Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel

Cédric Pasteur

THÈSE

PRÉSENTÉE A

L'UNIVERSITÉ PIERRE ET MARIE CURIE

ÉCOLE DOCTORALE :
École doctorale Informatique, Télécommunications et Électronique (Paris)

Par Cédric PASTEUR

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel

Directeur de recherche : Marc POUZET
Co-encadrant : Louis MANDEL

Soutenue le 26 novembre 2013

Devant la commission d'examen formée de :

Mme. Florence MARANINCHI	Professeur	Grenoble INP / ENSIMAG	Rapporteur
M. Jean-Bernard STEFANI	Directeur de recherche	INRIA	Rapporteur
M. Christian QUEINNEC	Professeur	UPMC	Président du jury
M. Gérard BERRY	Professeur	Collège de France	Examinateur
M. Didier RÉMY	Directeur de recherche	INRIA	Examinateur
M. Bruno PAGANO	Directeur scientifique	Esterel Technologies	Examinateur
M. Marc POUZET	Professeur	UPMC / ENS	Directeur de thèse
M. Louis MANDEL	Maître de conférence	Collège de France	Co-encadrant

Remerciements

Je tiens tout d'abord à remercier Florence Maraninchi et Jean-Bernard Stefani d'avoir accepté d'être mes rapporteurs. J'ai beaucoup apprécié mes discussions avec eux lors de ma visite à Grenoble. Elles m'ont permis de clarifier les points les plus complexes de mon manuscrit.

Je remercie également Christian Queinnec de m'avoir fait l'honneur de présider mon jury de thèse, ainsi que Gérard Berry, Didier Rémy et Bruno Pagano d'en avoir fait partie. C'est un grand honneur d'avoir dans mon jury des personnes dont les travaux sur ESTEREL et les types rangées ont inspiré le mien.

Je tiens à remercier tout particulièrement Marc Pouzet de m'avoir accueilli en stage puis d'avoir été mon directeur de thèse. Sa passion pour la programmation au sens noble du terme et pour la recherche de la simplicité m'ont guidé tout au long de cette thèse. Je tiens aussi à le remercier pour sa relecture attentive de mon manuscrit pendant ses vacances, qui m'a permis de grandement améliorer le document. Je m'excuse auprès de sa famille d'avoir ainsi empiété sur son temps de vacances.

Je souhaite également remercier Louis Mandel qui m'a encadré tout au long de ma thèse et a créé le merveilleux langage REACTIVEML sur lequel s'est porté mon travail. La qualité de ses travaux, en particulier de sa thèse, m'a permis de m'appuyer sur un socle solide pour mener à bien mes extensions du langage. Je veux également le remercier pour sa disponibilité et sa bonne humeur.

Je n'oublie pas tous les autres membres de l'équipe PARKAS et stagiaires avec qui j'ai passé ces trois dernières années, pour toutes les pauses café et bonbons de l'après-midi. Je tiens en particulier à remercier Timothy Bourke pour ses relectures attentives de mes articles même depuis l'autre bout du monde. Merci aussi à Léonard Gérard et Adrien Guatto avec qui j'ai écrit l'article sur mon travail de stage et participé à l'écriture du compilateur HEPTAGON. Je veux également remercier Mehdi Dogguy qui a écrit la boucle d'interaction en ligne de REACTIVEML que j'ai réutilisée pour mon extension du langage. Merci aussi à Albert Cohen qui a financé mes derniers mois de thèse sur un de ses projets.

Merci à Jean Vuillemin de m'avoir donné l'opportunité d'encadrer le projet de son cours *Système digital* à l'ENS, ainsi qu'à Emmanuelle Encrenaz et Karine Heydemann de m'avoir proposé d'animer les séances de TD/TP du cours MAREP à l'UPMC. Je garde un très bon souvenir de ces différentes expériences qui ont été très enrichissantes sur le plan personnel et professionnel. Je tiens à remercier Olivier Boudeville et Jingxuan Ma de EDF R&D pour les discussions sur SIMDIASCA qui ont motivé ce travail. Merci aussi à Frédéric Boussinot et Jean-Louis Giavitto qui ont participé à mon évaluation à mi-parcours et m'ont donné des conseils utiles pour la fin de ma thèse.

Enfin, je souhaite remercier mes amis et ma famille pour les moments de détente. J'ai une pensée toute particulière pour ma chérie Cécile que j'ai rencontré juste après le début de ma thèse et dont je n'ai pu me passer depuis. Merci aussi à Cécile et à ma famille pour l'organisation du pot après ma soutenance.

Table des matières

Première partie

Présentation informelle

CHAPITRE 1

Introduction

1.1 La concurrence dans les langages de programmation

Nous nous intéressons ici au problème de la *concurrence* dans les langages de programmation. Nous dirons qu'un système est concurrent s'il est constitué de comportements partiellement non ordonnés. Plus pragmatiquement, un système est concurrent lorsqu'il fait plusieurs choses « en même temps ». Par exemple, un métro doit, de façon concurrente, gérer sa vitesse, la signalisation, l'ouverture des différentes portes, la ventilation, l'éclairage, etc. Un navigateur internet est aussi un système concurrent, qui doit à la fois écouter les commandes de l'utilisateur par le clavier et la souris, gérer le rafraichissement de l'interface, la connexion avec les serveurs et le téléchargement des pages, etc.

Il s'agit donc bien d'une notion d'un modèle, et non d'implémentation. Il ne faut en particulier pas la confondre avec la notion de *parallélisme*. Le parallélisme concerne l'exécution du point de vue matériel de plusieurs fils d'exécution ou *threads* en simultané [1]. Le parallélisme est un des moyens pour implémenter la concurrence, mais il est également possible d'implémenter la concurrence de façon séquentielle. Les ordinateurs ont été capables d'effectuer plusieurs tâches de façon concurrente bien avant que les processeurs multi-cœurs n'apparaissent. Le but du parallélisme est d'exécuter les programmes plus rapidement en répartissant le travail entre les différentes unités de calcul. A l'opposé, nous chercherons ici à utiliser la concurrence comme moyen de simplifier l'écriture des programmes, puisque de nombreux problèmes s'expriment naturellement de façon concurrente. Nous parlerons tout de même de parallélisme dans ce manuscrit, mais il sera un moyen parmi d'autres pour exécuter les programmes et de façon transparente pour le programmeur. Il restera toujours possible d'exécuter les programmes de manière complètement séquentielle.

On trouve plusieurs approches traditionnelles pour programmer la concurrence :

Les threads systèmes Les systèmes d'exploitation fournissent un moyen de créer plusieurs fils d'exécution par programme, aussi appelés *processus légers* ou *threads*. Chaque thread dispose de son propre contexte, ce qui rend leur création et le passage d'un thread à l'autre coûteux. Ils peuvent accéder à un espace mémoire commun qu'ils utilisent pour communiquer et se synchroniser. Il est alors nécessaire d'utiliser des primitives d'exclusion mutuelle comme les verrous ou encore des instructions matérielles spéciales comme le *compare-and-swap* pour gérer les accès simultanés au même emplacement mémoire. Ce style de programmation est notoirement complexe. Il peut aboutir à des inter-blocages ou *deadlocks* et rend la mise au point des programmes difficile de par sa nature non déterministe et non reproductible. Dans [Lee06], E. Lee écrit un réquisitoire contre les threads en expliquant en détail ces problèmes.

La programmation événementielle L'idée de ce style de programmation, très utilisé pour la programmation d'interfaces graphiques, est de centrer le programme autour d'une *boucle*

1. Il n'y a pas de consensus dans la littérature sur l'utilisation de ces deux termes. Il sont parfois utilisés comme synonymes ou avec des sens inversés par rapport au choix que nous avons fait dans ce manuscrit.

d'événements. Chaque partie du programme enregistre des fonctions qui seront appelées lorsqu'un événement (comme un clic de souris) sera émis, ce qu'on appelle des *callbacks*. La boucle d'événements récolte les différents événements puis appelle les callbacks correspondants, qui vont à leur tour ajouter de nouveaux callbacks, et ainsi de suite. Cette approche de la concurrence permet de gérer efficacement de très nombreux comportements concurrents. Son gros inconvénient est que le code de chaque comportement est séparé en de nombreuses fonctions (une par événement) qui sont appelées par la boucle d'événements, ce que l'on appelle *l'inversion du contrôle*. Il devient donc difficile de comprendre la structure du programme, qui est éparpillé dans de nombreuses fonctions.

Les threads légers coopératifs (ou coroutines) Cette dernière approche tente de concilier le meilleur des deux approches précédentes. L'idée est de proposer un modèle de programmation direct, sans inversion du contrôle, mais sans la lourdeur des threads. Pour cela, on requiert des fils d'exécution qu'ils rendent régulièrement la main à l'ordonnanceur pour que les autres fils d'exécution puissent s'exécuter. C'est ce que l'on appelle *l'ordonnancement coopératif*. Cela permet une implémentation séquentielle efficace de la concurrence. C'est le modèle utilisé par des bibliothèques comme LWT [Vou08] en OCAML ou CONCURRENT HASKELL [JGF96].

Ces approches peuvent être utilisées dans tous les langages généralistes. Une autre alternative est d'intégrer la concurrence dans le langage. Cela permet à la fois de simplifier la vie du programmeur en proposant une syntaxe plus agréable et aussi d'offrir des garanties plus importantes. En effet, les bibliothèques de threads légers sont très souvent accompagnées de règles de bonne programmation à suivre pour obtenir un résultat correspondant à ses attentes. En intégrant la concurrence dans le langage, ces règles de bonne formation peuvent devenir des analyses statiques permettant de garantir la sûreté des programmes au moment de la compilation, de la même manière qu'un système de types évite de nombreuses erreurs. Les langages ADA [2] et ERLANG [3] sont par exemple des langages utilisés dans l'industrie qui intègrent la concurrence. Plusieurs langages généralistes font de même depuis peu, avec notamment les *calculs asynchrones* en F# [SPL11] et C#. Dans le monde académique, on peut citer notamment deux extensions de ML : CONCURENTML [Rep93] étend ML avec des primitives d'envoi de messages, alors que JOCAML [FLFMS04] s'inspire des constructions du *join-calculus* [FG02]. Nous allons nous intéresser dans ce manuscrit à une famille de langages intégrant la concurrence et le temps : les langages synchrones.

1.2 Les langages synchrones

Cadre

Les langages synchrones [BCE+03] sont des langages apparus au début des années 80 et dédiés à la programmation de systèmes embarqués (ce sont des *Domain Specific Languages* ou DSL). Par rapport aux langages traditionnellement utilisés dans ce domaine tels que C, les langages synchrones ont la particularité de mettre le *temps* et la *concurrence* au centre du langage. Ils reposent sur une sémantique définie formellement. Cela permet notamment d'utiliser des méthodes formelles pour la vérification automatique de propriétés des programmes. Les langages synchrones s'intéressent plus particulièrement aux systèmes suivants :

Réactifs Un système *réactif* interagit en permanence avec son environnement. Dans le cas d'un système embarqué, le programme va lire en continu les données des capteurs, puis calculer une commande appropriée pour les différents actionneurs. Cela est fait de façon périodique, le plus souvent en suivant une exécution dite *en boucle simple*, qui répète le cycle suivant :
- Attendre le déclenchement.
- Lire les entrées.
- Calculer la réaction.
- Écrire les sorties.

2. http://www.adaic.org/ada-resources/standards/ada05/
3. http://www.erlang.org/

D'autres exemples de systèmes réactifs sont les interfaces graphiques (ou GUI) et les jeux vidéos. On les distingue des systèmes dits *transformationnels* [HP85] qui n'interagissent avec l'environnement qu'au début du programme. C'est le cas par exemple d'un compilateur ou d'un logiciel de compression de fichiers.

Temps-Réel Les systèmes réactifs ont souvent des impératifs de temps-réel. Le temps d'exécution du programme est alors une propriété fonctionnelle du système, c'est-à-dire qu'il influe sur la correction du programme. Dans le cas d'un compilateur par exemple, le temps d'exécution est une variable que l'on cherchera à optimiser pour faciliter la vie de l'utilisateur, mais cela se fait de façon indépendante de la correction du programme. A l'opposé, un système embarqué confronté au monde physique doit respecter des contraintes strictes de temps de réponse. Un exemple typique est l'injection dans les moteurs à combustion, qui doit respecter un échéancier très précis.

Critiques Un système est dit critique lorsque des vies humaines dépendent de son bon fonctionnement. C'est par exemple le cas d'un avion, d'un train ou encore d'une centrale nucléaire. Il faut donc souvent pouvoir garantir avant l'exécution le temps de réaction du logiciel et pouvoir calculer le *temps d'exécution pire-cas* (*Worst-Case Execution Time* ou WCET) [WEE+08] d'un pas de la boucle d'interaction. En conséquence, l'expressivité des langages synchrones est volontairement réduite à des programmes statiques, sans aucune allocation dynamique de mémoire ni boucles non bornées. Ces langages ne sont donc pas Turing-complets.

Historiquement, on distingue deux grandes familles de langages synchrones :

- Les langages flot de données (*data-flow*) comme LUSTRE [CPHP87], SIGNAL [BLGJ91], LUCID SYNCHRONE [CP96] ou SCADE [4] dans lesquels on écrit des équations sur des suites infinies de valeurs appelées *flots*.
- Les langages impératifs comme ESTEREL [Ber97] ou QUARTZ [Sch09] qui proposent une programmation plus classique avec une notion de séquence.

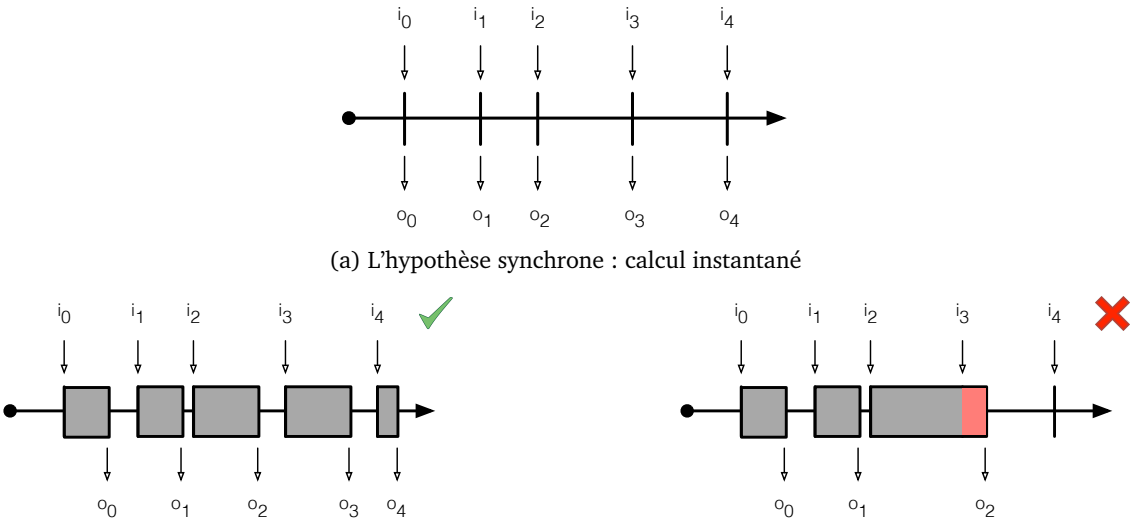
La frontière entre ces deux mondes s'est estompée dans les langages plus récents comme ESTEREL v7 ou SCADE 6. Nous allons dans la suite présenter plus en détail ces deux approches. Mais avant cela, nous allons présenter le point commun entre ces langages, c'est-à-dire le principe du synchrone.

L'hypothèse synchrone

L'approche synchrone [BCE+03] consiste à voir l'exécution d'un programme comme une succession d'instants logiques discrets. Dans un premier temps, on oublie donc totalement le temps réel : on ne parle que du premier instant, du deuxième instant, etc. L'autre point commun de ces langages est la présence d'un opérateur de composition parallèle synchrone, permettant de lancer de façon concurrente plusieurs processus. Il s'agit bien ici de concurrence logique, pas de parallélisme d'exécution. Cette concurrence dans le programme disparait généralement au moment de l'exécution, puisque le code généré est le plus souvent séquentiel. Comme pour le produit synchrone d'automates, la composition parallèle synchrone fait avancer les processus au même rythme, en *lock-step*. La notion d'instant logique est globale et partagée par tous les processus et permet donc une synchronisation implicite.

C'est pour faire le lien avec le temps-réel de l'exécution du programme que l'on va faire appel à l'hypothèse synchrone. L'idée est de supposer initialement que les calculs et communications au sein d'un instant sont instantanées, comme si l'on disposait d'un ordinateur infiniment rapide. Cela correspond à l'idée d'instant discret, de durée nulle. Pour chaque événement d'entrée i_k, on peut calculer instantanément la valeur de sortie o_k correspondante, comme le montre la figure 1.1a. Bien entendu, le temps de calcul n'est jamais nul au moment de l'exécution. Mais si on peut assurer que l'on calcule suffisamment vite, c'est-à-dire que la durée maximale du calcul est inférieure à la durée entre deux événements d'entrée (ou bien que les entrées varient de façon négligeable pendant la durée du calcul), comme sur la figure 1.1b, alors on peut effectivement faire comme si le calcul se déroulait de façon instantanée. En pratique, on calcule donc le WCET du programme généré et on vérifie ensuite s'il est compatible avec (c'est-à-dire plus petit que) la période d'exécution souhaitée.

4. http://www.esterel-technologies.com/products/scade-suite/



(a) L'hypothèse synchrone : calcul instantané

(b) Hypothèse vérifiée

(c) Hypothèse non vérifiée

FIGURE 1.1 – L'hypothèse synchrone

Les langages synchrones flot de données

Le principe des langages synchrones flot de données comme LUSTRE ou SIGNAL est d'écrire un ensemble d'équations définissant des suites infinies de valeurs indicées par les instants logiques, appelées *flots*. Voici un exemple simple en LUSTRE de fonction sur les flots, ou *nœud*, appelée `event_counter` qui prend en entrée un flot de booléens `tick` et qui retourne un flot d'entiers `cpt` égal au nombre de fois où le flot d'entrée a été vrai jusqu'à maintenant : [5]

```
node event_counter(tick:bool) returns (cpt:int);
var pcpt:int;
let
  cpt = if tick then pcpt + 1 else pcpt;
  pcpt = 0 -> pre cpt;
tel
```

Les équations sur les flots sont implicitement quantifiées universellement sur les instants. Cela signifie que les opérations de base comme l'addition ou le test sont effectuées point-à-point. Ainsi, la première équation doit être comprise comme :

$$\forall i \in \mathbb{N}.\ cpt_i = \texttt{if}\ tick_i\ \texttt{then}\ pcpt_i + 1\ \texttt{else}\ pcpt_i$$

La seconde équation utilise l'opérateur d'initialisation -> et l'opérateur de délai non initialisé **pre**. `pcpt` est donc égal à `0` au premier instant, puis à la valeur précédente de `cpt` aux instant suivants. Cette équation est donc équivalente à :

$$pcpt_0 = 0$$
$$\forall i \in \mathbb{N}.\ pcpt_{i+1} = cpt_i$$

On peut également représenter les différentes valeurs des flots au cours du temps dans un *chronogramme*, c'est-à-dire un tableau où chaque ligne correspond à un flot et chaque colonne correspond à un instant du programme :

5. La syntaxe peut porter à confusion puisque le point-virgule représente la composition parallèle synchrone.

tick	*tt*	*ff*	*tt*	*ff*	...
cpt	1	1	2	2	...
pre cpt	nil	1	1	2	...
pcpt	0	1	1	2	...

La valeur `nil` représente une valeur non initialisée, puisque la valeur précédente d'un flot n'est pas définie au premier instant.

Un point à signaler dans le programme précédent est que tous les flots sont calculés à tous les instants. En particulier, le **if** évalue la condition et les deux branches à tous les instants puis choisit la bonne valeur, comme un multiplexeur dans un circuit numérique. Pour qu'un flot ne soit calculé qu'à certains instants, ce qu'on appelle du *sous-échantillonnage*, on peut utiliser l'opérateur de filtrage **when** : `e when c` n'est présent que lorsque le flot `c` vaut *tt*, auquel cas il est égal à la valeur de `e`. Il est absent, et donc n'est pas évalué, lorsque la condition `c` est fausse. Réciproquement, l'opérateur **current** prend un flot qui n'est pas présent à tous les instants et renvoie un flot présent à tous les instants, égal à la dernière valeur présente de l'entrée. On peut utiliser ces deux opérateurs pour définir une autre version du compteur d'événements, qui ne fait l'addition que lorsque l'événement est présent :

```
node event_counter2(tick:bool) returns (cpt:int);
var pcpt:int when tick;
let
  cpt = current (pcpt);
  pcpt = (1 when tick) -> pre pcpt + (1 when tick);
tel
```

`pcpt` est maintenant un compteur sans test, qui n'est activé que lorsque `tick` est vrai (la notation `.` représente l'absence de valeurs) :

tick	*tt*	*ff*	*tt*	*ff*	...
cpt	1	1	2	2	...
pcpt	1	.	2	.	...

Puisque les flots ne sont pas nécessairement présents à tous les instants, il faut garantir que l'on ne va jamais chercher à lire un flot absent. Cette vérification est faite par une analyse du compilateur appelée *calcul d'horloges* [Cas92]. Elle assigne à chaque flot son *horloge*, qui est un flot booléen vrai lorsque le flot est présent et faux sinon. Par exemple, l'horloge de `pcpt` est `tick`, ce que l'on peut voir dans la déclaration de la variable. On vérifie ensuite que les horloges des différents flots sont correctes. Par exemple, deux flots additionnés doivent avoir la même horloge. C'est pour cela que l'on a dû filtrer la constante `1` avec `tick`. [CP03] montre qu'une version simplifiée de cette analyse peut s'exprimer comme un système de types à la ML [DM82] où les horloges sont des types abstraits [LO92]. Cet article se base sur le langage LUCID SYNCHRONE, qui est un langage synchrone flot de données reprenant certains traits des langages fonctionnels comme l'inférence de types et l'ordre supérieur.

Les langages synchrones impératifs

Le premier langage synchrone impératif est le langage ESTEREL. Il reprend des concepts plus proches des langages impératifs comme C, avec la séquence ou les boucles. La grande différence avec ces langages est que l'on dispose d'un opérateur de composition parallèle synchrone noté `||` et d'une notion d'instant logique global. La communication entre les processus se fait à l'aide de *signaux* avec diffusion instantanée (*instantaneous broadcast*). Voici l'exemple du programme ESTEREL le plus célèbre appelé ABRO et tiré de [Ber97]. Il émet sa sortie `O` une fois qu'il a reçu ses deux entrées `A` et `B`. La troisième entrée `R` permet de remettre le processus dans son état initial :

```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```

L'émission d'un signal se fait par un appel à `emit`. Tous les autres processus voient alors la présence du signal pendant l'instant courant. Contrairement à LUSTRE, il existe une dissymétrie puisqu'un signal est considéré par défaut absent s'il n'est pas émis. `await A` termine dans l'instant où le signal `A` est présent. Comme la composition synchrone termine lorsque les deux branches ont terminé, l'émission de `O` ne se fait que lorsque les deux entrées ont été émises sans réinitialisation.

Le langage offre également la possibilité de tester la présence d'un signal avec la construction `present S then p1 else p2 end` qui exécute instantanément `p1` (resp. `p2`) si le signal `s` est présent (resp. absent). Cette possibilité de réagir instantanément à l'absence d'un signal a des conséquences non triviales, comme le montre l'exemple suivant :

```
signal S in
  present S then nothing else emit S end
end
```

Si le signal `S` est absent, il est émis et devient donc présent. Ce programme doit être rejeté puisqu'il révèle une incohérence. On peut également écrire des programmes non-déterministes :

```
signal S in
  present S then emit S else nothing end
end
```

Dans cet exemple, on peut supposer que `S` est présent ou absent et les deux hypothèses sont vérifiées. Là aussi, ce programme doit être rejeté puisque l'on doit éviter le non-déterminisme dans le cas d'un système critique. D'autres exemples de programmes montrent des cas où l'on peut trouver un et un seul statut par signal, mais qui ne correspondent à l'intuition de causalité de la séquence. Cela signifie que l'émission d'un signal doit précéder (au sens de la séquence) le choix de sa présence ou son absence :

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

De nombreux travaux ont été menés sur la définition de la classe des programmes ESTEREL « raisonnables » que l'on veut accepter et sur la définition d'une analyse permettant de détecter les programmes qui ne rentrent pas dans cette classe. Ils ont abouti à la définition d'une sémantique *constructive* du langage [Ber96], inspirée de la propagation des signaux dans un circuit électrique.

Le synchrone réactif à la REACTIVEC : au-delà des systèmes embarqués

Confronté aux problèmes de causalité d'ESTEREL, [Bou91] propose une solution alternative consistant à réduire les possibilités du langage pour obtenir des processus causaux « par construction ». La principale restriction est de ne pouvoir réagir à l'absence d'un signal qu'à l'instant suivant. Ainsi, `present s then p1 else p2` exécute `p1` instantanément si le signal est présent, mais exécute `p2` à l'instant suivant si le signal est absent. Tous les programmes précédents ont donc maintenant une sémantique bien définie : [6]

```
signal s in
present s then () else emit s
```

6. On reprend ici la syntaxe de REACTIVEML que l'on présentera dans le chapitre 2.

Dans cet exemple, le signal s est absent au premier instant puisqu'il n'est pas émis. On exécute donc la branche **else** au second instant, ce qui signifie que s est présent au second instant.

L'avantage de ne plus avoir à faire d'analyse de causalité est que l'on peut plonger le langage dans un langage existant (on parle alors d'*Embedded DSL*). Dans [Bou91], des constructions synchrones inspirées d'ESTEREL sont ajoutées au-dessus du langage C pour former REACTIVEC. Cette approche permet d'utiliser un langage généraliste pour les structures de données et les fonctions instantanées, auquel on ajoute une sur-couche synchrone. Elle a ensuite été appliquée notamment à JAVA pour les SUGARCUBES [BS98] et JUNIOR [HSB99].

L'absence de problèmes de causalité rend immédiate la prise en compte de la création dynamique de processus. On sort alors du cadre des systèmes embarqués critiques, dont on oublie toutes les contraintes, pour s'attacher au problème plus général de la concurrence dans les langages de programmation généralistes. Les applications vont des interfaces graphiques, comme les ICOBJS [BSTH01], à la simulation discrète.

Le langage REACTIVEML [MP05, Man06], auquel nous nous intéressons dans ce manuscrit, reprend ces idées en étendant un noyau fonctionnel à la ML (plus précisément un sous-ensemble d'OCAML) avec des constructions synchrones. Par rapport à ses prédécesseurs, il dispose d'une formalisation le rapprochant de la littérature sur les langages fonctionnels (sémantique opérationnelle, typage). De plus, l'approche fonctionnelle avec ordre supérieur permet d'obtenir un langage avec une très grande expressivité et une concision dans l'écriture des programmes. REACTIVEML a été utilisé notamment pour la simulation de réseaux ad-hoc [MB05], de réseaux de capteurs [SMMM06], pour un interprète du langage synchrone DSL [ABMS11] et du langage de suivi de partition ANTESCOFO [BJMP13, Con08]. Le compilateur REACTIVEML [7] est implémenté en OCAML et génère du code OCAML séquentiel, qui est lié à un moteur d'exécution lui-aussi écrit en OCAML. Il dispose également d'une boucle d'interaction (ou *read-eval-print loop*) permettant une programmation interactive [MP08a], qui est aussi disponible en ligne [8]. Nous reviendrons plus en détail sur le langage dans le chapitre 2 qui présente une sémantique informelle du langage à partir d'exemples.

Distribution et parallélisation de code synchrone

Les premiers travaux sur la génération de code distribué pour les langages synchrones ont été motivés non pas par souci de performance mais par souci de distribution physique du programme sur plusieurs microprocesseurs ou processeurs spécialisés (DSP). Par exemple, dans un système embarqué, un processeur spécialisé peut être en charge des tâches intensives (comme le modulation d'un signal radio ou la compression d'une vidéo) alors qu'un processeur généraliste se charge du reste du programme. L'objectif recherché est de pouvoir programmer ce type de systèmes distribués physiquement avec un seul programme.

La première approche consiste à partir d'un programme synchrone puis à chercher une manière de le répartir tout en gardant la sémantique synchrone. [Gir05] dresse un panorama des méthodes existantes dans ce domaine. La seconde approche consiste à étendre le modèle synchrone pour modéliser le mélange synchrone/asynchrone des systèmes distribués. L'extension la plus connue du modèle synchrone est la notion d'architecture GALS, pour *Globally Asynchronous Locally Synchronous*. Son principe, inspiré par le monde des circuits numériques, est de voir un programme comme un ensemble d'îlots synchrones communiquant de façon asynchrone. C'est notamment le modèle de l'outil de *co-design* POLIS [Bal97], de l'environnement de simulation PTOLEMY [BHLM94], des *Communicating Reactive Processes* ou CRP [BRS93] qui étend ESTEREL, ou encore des langages SYSTEMJ [MSRG10] et DSYSTEMJ [MGS10] implémentés au-dessus de JAVA. La communication entre îlots se fait le plus souvent par rendez-vous bloquant, suivant le modèle de CSP [Hoa78].

Le langage flot de données SIGNAL permet également de spécifier des GALS, qui pourront être exécutés en parallèle dans plusieurs threads ou dans un contexte distribué [GG10, LGTLL03]. En effet, le langage permet de décrire les liens entre les horloges non seulement par inclusion, comme

7. http://reactiveml.org
8. http://reactiveml.org/tryrml/tryrml.html

pour le filtrage en LUSTRE, mais aussi par des relations (au sens mathématique). Cela permet de décrire des systèmes ayant plusieurs horloges globales, ce qui correspond à un GALS. Nous reviendrons plus en détail sur les possibilités de SIGNAL dans le chapitre 5.

Un modèle proche des GALS a été utilisé pour la parallélisation des langages synchrones impératifs. Les FAIRTHREADS [Bou06, SBS04] et le langage FUNLOFT [BD08] proposent en effet un modèle mélangeant ordonnancement coopératif au sein des îlots synchrones et ordonnancement préemptif entre îlots. La communication se fait cette fois en liant deux îlots, c'est-à-dire en synchronisant leurs instants afin que les processus qu'ils contiennent puissent communiquer. Le langage DSL [ABMS11] propose un modèle proche, mais où les communications se font par migration de processus d'un îlot à l'autre. Une approche alternative [CGP12] pour la parallélisation de langages synchrones flot de données est basée sur la notion de *futur* : un futur est une promesse de résultat, que l'on ne lit que lorsqu'on a réellement besoin de la valeur. Le calcul correspondant peut donc se faire dans un thread séparé.

1.3 Présentation du travail

Motivations

La motivation de cette thèse est le domaine de la simulation discrète, qui est le principal domaine d'application de REACTIVEML. Plus particulièrement, elle prend son origine dans une collaboration avec l'équipe de EDF R&D développant SIMDIASCA[9] (pour *Simulation of Discrete Systems of All Scales*), un framework de simulation discrète écrit en ERLANG. Il a été conçu afin de simuler des réseaux de compteurs électriques intelligents, avec pour objectif de parvenir à simuler des réseaux de très grande taille, contenant des centaines de milliers voire des millions d'agents. Ce framework utilise une approche synchrone qui permet de garantir simplement l'ordre entre les événements, ce qui est à l'origine de la collaboration.

L'écriture de tels programmes pose deux questions :

Sait-on les programmer simplement et de façon modulaire ?
Sait-on atteindre de tels ordres de grandeur tout en conservant une exécution reproductible ?

Les exemples existants de simulation de réseaux ad-hoc [SMMM06, MB05] montrent que REACTIVEML est adapté pour ce domaine. Il se pose toutefois plusieurs questions auxquelles nous allons tenter de répondre dans ce manuscrit :

La gestion de multiples échelles de temps et du *raffinement temporel*.
La parallélisation et la distribution de programmes REACTIVEML.

Échelles de temps et raffinement temporel

Le domaine de la simulation discrète pose la question de la programmation de systèmes synchrones avec plusieurs échelles de temps. Dans l'exemple de la simulation d'un réseau de capteurs, on trouve deux échelles de temps distinctes : l'échelle de temps pour la modélisation du logiciel interne des capteurs et des communications entre capteurs, où le temps se compte en millisecondes, et l'échelle de temps pour la modélisation du fonctionnement de la radio, où l'on parle plutôt de microsecondes. En outre, on souhaiterait pouvoir modifier la précision de la simulation en remplaçant une modélisation précise mais coûteuse d'un système, qui prend plusieurs instants à être simulée, par une approximation instantanée. Cela doit être possible sans que le comportement observable du système ne change du point de vue des autres composants du système. Dans un langage synchrone, on s'intéresse tout particulièrement au nombre d'instants qui s'écoulent entre deux actions, par exemple entre l'émission sur un signal d'entrée et l'émission sur le signal de sortie d'un processus. Dans ce document, nous appellerons *raffinement temporel* cette possibilité de remplacer un processus par un autre sans modifier le comportement observable.

9. http://innovation.edf.com/recherche-et-communaute-scientifique/logiciels/sim-diasca-80703.html

(a) Avant (b) Après

FIGURE 1.2 – Le raffinement temporel avec le sous-échantillonnage (les traits verticaux représentent les instants d'une horloge, les lignes horizontales représentent des processus lancés en parallèle)

Nous ne chercherons pas ici à définir formellement les notions de raffinement ou d'équivalence de programmes, comme cela est par exemple fait en LUSTRE dans [MC05].

Cette notion de raffinement est une des motivations de SYSTEMC [10] [Pan01]. Il s'agit d'une bibliothèque C++ pour la conception de logiciels et circuits. Elle est centrée autour d'un moteur de simulation à événements discrets et permet de décrire le comportement fonctionnel et temporel d'un système de façon concurrente. Le choix d'un langage généraliste, contrairement aux langages de description de circuits habituels comme VHDL ou VERILOG, permet de gérer différents niveaux d'abstraction. On peut ainsi partir d'une description fonctionnelle d'un système, utilisant toutes les possibilités de C++, puis la raffiner progressivement jusqu'à une description précise d'un circuit au bit près. En outre, la notion de δ-cycle, c'est-à-dire la possibilité d'exécuter plusieurs pas de calcul sans que le temps de la simulation n'avance, simplifie le raffinement.

Le raffinement n'est pas immédiat dans un modèle synchrone : il est clair qu'on ne peut pas remplacer un processus qui calcule son résultat instantanément par un autre prenant trois instants sans changer a priori le comportement global du programme. La réponse traditionnelle à ce problème est la notion de sous-échantillonnage : on crée une nouvelle échelle de temps en prenant un sous-ensemble des instants d'une autre échelle de temps. On a vu par exemple que le sous-échantillonnage se faisait en LUSTRE avec l'opérateur **when** de filtrage. Cette approche permet effectivement d'obtenir des processus évoluant à des rythmes différents, mais elle pose de graves problèmes de modularité. Supposons que l'on se trouve dans la situation de la figure 1.2a, avec un seul processus activé à tous les instants. Si un processus souhaite créer une échelle de temps plus rapide, il doit en fait ralentir le premier processus, en le sous-échantillonnant par rapport au rythme global du programme. C'est ce que l'on obtient sur la figure 1.2b. Le sous-échantillonnage permet donc de faire un raffinement temporel, mais il ne permet pas de faire simplement plusieurs raffinements en parallèle, sur des rythmes différents.

Plus récemment, une alternative au sous-échantillonnage appelée *raffinement d'horloges* [GBS10b] a été proposée dans le langage QUARTZ [Sch09]. L'idée est de créer une nouvelle échelle de temps en subdivisant les instants. Nous allons dans ce manuscrit suivre la même approche. Nous détaillerons les similitudes et les différences avec l'approche de Gemünde et al. dans la partie 5.3. Le pendant de cette idée dans les langages flot de données est la notion d'*horloges entières* [Gér08, Pla10], qui représentent la possibilité de consommer et produire plusieurs valeurs par instant sur un flot. Elles peuvent également traduire des *salves* d'instants, c'est-à-dire des séquences de calculs inobservables. Le développement d'un langage synchrone flot de données basé sur ces principes est en cours [GM13].

Parallélisation de REACTIVEML

Dans le cas de SIMDIASCA, on souhaite simuler des systèmes contenant des centaines de milliers voire des millions d'agents. Pour arriver à de tels ordres de grandeur, il est nécessaire de passer par une exécution parallèle ou distribuée du code généré. Le code séquentiel généré par le compilateur REACTIVEML est très efficace. Malheureusement, les performances séquentielles des microprocesseurs ont atteint un plateau et les évolutions actuelles vont vers la multiplication des cœurs dans les microprocesseurs, qu'il faudra bien un jour mettre à profit. Il faut bien insister sur le

10. http://www.systemc.org/home/

fait que l'on cherche à paralléliser l'exécution d'un seul programme et pas à proposer un modèle de programmation de systèmes distribués. Les réponses que l'on va apporter sont donc adaptées à ce premier cadre et pas nécessairement au second.

La parallélisation efficace d'un programme synchrone passe en grande partie par la suppression des synchronisations inutiles. En effet, dans un monde synchrone, les instants sont globaux et partagés par tous les processus. Tous les processus se synchronisent à la fin de chaque instant, qu'ils communiquent ou non. Le but est de limiter au maximum ces synchronisations inutiles afin d'améliorer les performances. On souhaite tout de même conserver les propriétés de déterminisme et de reproductibilité du langage, qui sont capitales dans le cadre de la simulation discrète.

Les contributions de cette thèse

La contribution principale de cette thèse est la définition d'une extension du modèle synchrone que nous avons appelée *domaines réactifs* et qui a été publiée dans [MP13b]. Contrairement au sous-échantillonnage qui crée une nouvelle échelle de temps en allant plus lentement, c'est-à-dire en sélectionnant un sous-ensemble des instants, un domaine réactif crée une nouvelle échelle de temps en allant plus vite, en subdivisant les instants. Pour cela, le domaine réactif crée une notion d'instant local, inobservable depuis l'extérieur du domaine. Les apports de cette nouvelle construction sont :

Modularité L'ajout des domaines réactifs permet une véritable modularité dans le modèle synchrone. La possibilité de masquer des instants de façon modulaire facilite en particulier le raffinement temporel.

Expressivité L'extension du langage permet de simplifier l'écriture de nombreux programmes. Elle permet par exemple d'éviter de maintenir à la main la valeur des signaux et simplifie la synchronisation entre processus.

Reproductibilité Les domaines réactifs permettent de rester dans un monde synchrone et donc de conserver les propriétés de déterminisme et de reproductibilité.

Simplicité d'implémentation La notion de domaine réactif est en quelque sorte une *réification* [FW84] du moteur d'exécution de REACTIVEML. Un domaine réactif est comme un programme synchrone à l'intérieur d'un autre programme synchrone, mais qui peut communiquer avec son hôte. C'est pourquoi l'implémentation des domaines réactifs est aisée, puisque l'on a déjà toutes les briques de base.

Parallélisation L'ajout d'instants locaux permet de désynchroniser partiellement les programmes et de distinguer ce qui est de l'ordre de la communication locale et ce qui concerne une synchronisation globale. Cela permet donc de répondre à notre problématique d'implémentation parallèle et distribuée efficace de REACTIVEML.

L'introduction de multiples échelles de temps dans le langage ajoute des contraintes sur l'écriture des programmes. Nous définirons plusieurs systèmes de *types-et-effets* [LG88] pour garantir la sûreté des programmes en présence de domaines réactifs. Le premier permet de garantir une utilisation correcte des signaux dans le langage étendu. Nous l'appellerons *calcul d'horloges* puisqu'il reprend les objectifs et les techniques de cette analyse dans les langages flot de données. Le second est une analyse statique appelée *analyse de réactivité* qui garantit que le programme ne va pas entrer dans une boucle instantanée qui empêche les instants logiques de progresser. Elle a été publiée en français dans [MP13a] et une version étendue est en cours de soumission [MPP13]. Nous montrerons également une approche nouvelle du *subeffecting* [NNH99], c'est-à-dire du sous-typage restreint aux effets, qui utilise la notion de types rangées [Rém94].

La seconde contribution de cette thèse est un moteur d'exécution parallèle et distribuée de REACTIVEML. Nous montrerons plus précisément deux approches :

– Avec *threads* et communication par mémoire partagée. Cette expérience a été faite dans le langage F# [11], à cause des limitations du langage OCAML qui ne permet pas l'exécution de

11. http://tinyurl.com/fsspec

plusieurs threads en parallèle. Elle utilise des techniques traditionnelles de l'ordonnancement de tâches comme le *vol de tâches* (ou *work stealing*) [HS08].
– Avec des processus (lourds) et communication par envoi de messages. Ce moteur est implémenté en OCAML et utilise des processus système communiquant par envoi de messages à l'aide de la bibliothèque MPI [12].

Enfin, j'ai également participé pendant cette thèse au développement du compilateur du langage HEPTAGON, qui est un langage synchrone flot de données avec des automates de mode hiérarchiques [MR98, CPP05]. J'ai en particulier continué le développement d'une méthode d'optimisation mémoire pour le langage, que j'avais commencé pendant mon stage de recherche [Pas10]. Le but de cette optimisation est de limiter les copies et l'utilisation de la mémoire en présence de tableaux. Nous présenterons dans l'annexe B ces travaux [GGPP12], qui ont obtenu le prix du meilleur article de la conférence LCTES'12.

Plan du document

La première moitié du document est consacrée aux domaines réactifs. Puisque ce manuscrit traite de programmation, nous nous appuierons sur de nombreux exemples pour présenter l'approche actuelle, ses limites et les solutions que nous proposons. La première partie du manuscrit présente donc de façon informelle la notion de domaine réactif. Nous commencerons d'abord dans le chapitre 2 par décrire plus en détail le langage REACTIVEML. Nous montrerons également plusieurs exemples plus complexes pour motiver les extensions du langage. Au passage, nous présenterons une première extension : les *signaux à mémoire*. Ensuite, nous présenterons dans le chapitre 3 la notion de domaine réactif, en nous appuyant là encore sur des exemples.

Nous donnerons ensuite dans la seconde partie plusieurs sémantiques formelles du langage étendu, en les démontrant équivalentes. Le chapitre 4 présente la sémantique à grands pas ou *comportementale* du langage. Cela nous permettra de discuter plus précisément des choix effectués. Le chapitre 5 présente la sémantique à petits pas ou *opérationnelle* du langage. Elle donne une vision plus opérationnelle du langage, c'est-à-dire plus proche de l'implémentation. Nous montrerons son équivalence avec la sémantique comportementale. Nous comparerons également notre approche avec les travaux similaires du domaine.

La partie suivante traite de systèmes de types pour garantir la sûreté des programmes. Le chapitre 6 présente le calcul d'horloges, qui permet de garantir une utilisation sûre des signaux dans le cadre étendu avec plusieurs échelles de temps. Nous montrerons la correction de cette analyse vis-à-vis de la sémantique à petits pas, c'est-à-dire qu'un programme bien typé ne peut pas avoir d'erreurs à l'exécution. Nous utiliserons un autre système de types-et-effets dans le chapitre 7 pour implémenter une analyse de réactivité. Cette analyse garantit que le programme ne va pas entrer dans une boucle instantanée qui empêche les instants logiques de progresser. Nous montrerons sa correction en utilisant cette fois la sémantique à grands pas. Le chapitre 8 traite de plusieurs extensions des deux systèmes de types, avec en particulier une présentation nouvelle du *subeffecting* inspirée des types rangées.

La dernière partie du manuscrit s'intéresse à l'implémentation et aux applications. Le chapitre 9 présente tout d'abord l'implémentation séquentielle des domaines réactifs. Il décrit également plusieurs améliorations du moteur d'exécution de REACTIVEML. Nous parlerons ensuite d'une implémentation parallèle de REACTIVEML basée sur le vol de tâches et utilisant des threads et des communications par mémoire partagée. Nous décrirons cette implémentation dans le cas de REACTIVEML classique, puis nous l'étendrons aux domaines réactifs. Nous décrirons également un moteur d'exécution parallèle et distribuée de REACTIVEML avec domaines, basé sur l'envoi de messages. Nous montrerons des résultats expérimentaux sur l'efficacité des différents moteurs d'exécution. Le chapitre 10 présente une application des domaines réactifs. Il s'agit d'écrire un interprète d'ESTEREL en utilisant les nouvelles constructions du langage. La difficulté réside dans la décision de l'absence des signaux, qui se fait par itérations successives en utilisant les techniques de [Ber96]. Nous obtiendrons au final un interprète fidèle, puisque chaque instant du programme

12. http://www.mcs.anl.gov/research/projects/mpi/

ESTEREL correspond à un instant de l'interprète. La seconde partie de ce chapitre étend cet interprète pour gérer une hypothétique extension d'ESTEREL avec des domaines réactifs. Cela nous permettra de discuter les différents choix faits au moment de la définition des domaines réactifs et de voir si d'autres choix auraient été possibles dans un contexte plus statique.

Nous terminerons enfin par le chapitre 11 en présentant plusieurs extensions et pistes de recherche, avant de conclure. L'annexe A présente un encodage partiel en REACTIVEML classique des différentes nouveautés introduites dans le manuscrit. Sur un tout autre sujet, l'annexe B reprend l'article [GGPP12] sur une optimisation mémoire pour les langages synchrones flot de données.

Notations

Dans la suite du manuscrit, nous utiliserons de nombreux exemples pour expliquer et illustrer le langage et les différentes nouveautés. Nous incitons le lecteur à essayer ces exemples qui sont fournis avec le compilateur à l'adresse suivante :

http://reactiveml.org/these_pasteur/rml-these.zip

Les instructions pour installer le compilateur et compiler les exemples sont incluses dans l'archive. Il est également possible de les essayer dans la boucle d'interaction du langage, disponible en ligne à l'adresse : http://reactiveml.org/these_pasteur/tryrpml/. Il s'agit d'une adaptation à REACTIVEML du site TryOCaml faite par Mehdi Dogguy. Elle est basée sur l'outil `js_of_ocaml`, [13] qui permet de compiler du bytecode OCAML vers JAVASCRIPT.

Nous appellerons REACTIVEML classique la version du langage antérieure à la thèse, sans les extensions que nous allons définir dans la suite, c'est-à-dire les signaux à mémoire et les domaines réactifs. Nous avons fait dans ce manuscrit plusieurs choix de syntaxe un peu différents. Les exemples présentés ici ne sont donc pas forcément acceptés tels quels par le compilateur REACTIVEML classique.

Nous ajouterons aux extraits de programme REACTIVEML le nom du fichier correspondant dans cette archive :

@ exemples/ch1/hello_world.rml

```
let process main =
  print_endline "Hello world!"
```

Cela indique que le code source complet est disponible dans l'archive avec le compilateur. Le symbole @ à coté du nom du fichier indique que l'on peut cliquer sur ce nom pour accéder directement au code complet. Le programme peut être compilé avec les commandes suivantes :

```
> cd exemples/ch1
> make hello_world.rml.native
> ./hello_world.rml.native
Hello world!
```

Les lignes précédés de '>' indiquent des commandes à entrer dans le terminal, alors que les autres sont les sorties du programme. La sortie du compilateur est affichée comme :

```
val main : unit process
```

Par défaut, les programmes compilés s'exécutent jusqu'à leur terminaison à vitesse maximale. Pour tester les programmes, il peut être utile de choisir le nombre d'instants exécutés avec l'option `-n` et la fréquence d'échantillonnage avec l'option `-sampling`. Par exemple :

```
> ./hello_world.rml.native -n 3 -sampling 0.2
```

exécute les trois premiers instants du programme `hello_world.rml.native` toutes les $0,2$ secondes.

13. http://ocsigen.org/js_of_ocaml/

CHAPITRE 2

Un cours accéléré de ReactiveML

Le langage REACTIVEML[1] a été créé par Louis Mandel pendant sa thèse [Man06]. Il s'agit d'une extension réactive de ML : le noyau du langage est un langage fonctionnel d'ordre supérieur, auquel on ajoute des constructions synchrones inspirées du modèle synchrone réactif de REACTIVEC [Bou91]. L'exécution d'un programme REACTIVEML est donc une succession d'instants logiques. On dispose également de la composition parallèle synchrone, qui permet de lancer de façon concurrente plusieurs processus communiquant à l'aide de signaux à diffusion instantanée.

2.1 Introduction à REACTIVEML

Instants et parallélisme synchrone Puisque REACTIVEML est une extension de ML, tout programme ML est aussi un programme REACTIVEML. Dans l'exemple suivant, on définit le type algébrique des arbres binaires et un combinateur pour itérer une fonction `f` sur un arbre avec l'ordre préfixe :

exemples/ch2/preorder.rml

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree

let rec preorder f t = match t with
| Empty -> ()
| Node(l, v, r) ->
   f v; preorder f l; preorder f r
```

```
val preorder : ('a -> 'b) -> 'a tree -> unit
```

La syntaxe concrète du langage est celle d'OCAML, langage sur lequel s'appuie REACTIVEML. On commence par définir le type algébrique `'a tree` des arbres binaires dont les nœuds contiennent une étiquette de type `'a`. Un arbre est soit vide, soit formé d'un fils gauche, d'une étiquette et d'un fils droit. Le parcours préfixe de l'arbre est une fonction récursive définie par filtrage. Si l'arbre est vide, alors on ne fait rien. Sinon, on applique la fonction `f` à l'étiquette du nœud, puis on parcourt récursivement le fils gauche puis le fils droit. Notre premier exemple de programme REACTIVEML montre que l'on peut programmer aussi simplement le parcours en largeur de l'arbre :

exemples/ch2/levelorder.rml

```
let rec process levelorder f t = match t with
| Empty -> ()
| Node (l, v, r) ->
    f v; pause;
    (run levelorder f l || run levelorder f r)
```

```
val levelorder : ('a -> 'b) -> 'a tree -> unit process
```

1. http://reactiveml.org

On définit ici un *processus* `levelorder`, prenant en argument une fonction `f` et un arbre `t`. Contrairement aux fonctions ML dont l'exécution est instantanée, celle d'un processus peut durer plusieurs instants logiques. Le langage impose une séparation stricte entre les fonctions ML et les processus qui peuvent seuls avoir un comportement réactif. C'est pourquoi on utilise le mot-clé **process** pour définir un processus. Le parcours de l'arbre utilise les instants logiques pour la synchronisation. Pour cela, le processus attend l'instant suivant avec l'opérateur **pause**, puis lance récursivement et en parallèle le parcours sur les fils gauche et droit. On doit comprendre **run** `levelorder f l` comme **run** `((levelorder f) l)`. Puisque tous les processus partagent la même notion d'instant, les parcours des fils gauche et droit avancent au même rythme, progressant d'un étage par instant.

Signaux Les processus lancés en parallèle communiquent à l'aide de signaux à diffusion instantanée. Cela signifie que tous les processus ont une vision cohérente d'un signal au cours d'un instant : il est soit présent, soit absent, mais son statut ne peut pas varier pendant l'instant. Voici un premier exemple d'utilisation d'un signal :

exemples/ch2/signal_present.rml
```
let process signal_present =
  signal s in
  emit s
  ||
  present s then print_endline "Present !" else print_endline "Absent !"
```

Le mot-clé **signal** permet de déclarer un nouveau signal local. **signal** `s` **in** `e1 || e2` se lit comme **signal** `s` **in** `(e1 || e2)`. Ensuite, la première branche émet le signal `s`, c'est-à-dire qu'il devient présent au cours de cet instant. Un signal est absent s'il n'est pas émis. On peut ensuite tester la présence du signal à l'aide de la construction **present**/**then**/**else**, qui va dans cet exemple exécuter la branche **then** :

```
> ./signal_present.rml.native
Present !
```

Supposons maintenant que le signal ne soit pas émis :

exemples/ch2/signal_absent.rml
```
let process signal_absent =
  signal s in
  present s then print_endline "Present !" else print_endline "Absent !"
```

Sans surprise, le programme exécutera maintenant la seconde branche du **present** :

```
> ./signal_absent.rml.native
Absent !
```

Mais ce que l'on ne voit pas dans la sortie du programme, c'est que cet affichage a lieu au cours du second instant du programme. En effet, REACTIVEML reprend les restrictions du modèle de REACTIVEC [Bou91]. Cela signifie donc que dans le cas de l'expression **present** `s` **then** `e1` **else** `e2`, `e1` est exécutée immédiatement si le signal est présent, mais que `e2` n'est exécutée qu'à l'instant suivant si le signal est absent. On peut également attendre l'émission d'un signal à l'aide de la construction **await immediate** `s`, qui termine instantanément lorsque le signal `s` est émis. Cela nous permet par exemple de créer un détecteur de front montant `edge`, qui écoute un signal `s_in` et émet le signal `s_out` lorsque `s_in` est présent alors qu'il était absent à l'instant précédent. On utilise ici une boucle inconditionnelle qui s'écrit **loop**/**end** :

```
let process edge s_in s_out =
  loop
    present s_in then pause
    else (await immediate s_in; emit s_out)
  end
```

Remarque 1. La construction **await immediate** peut aussi s'exprimer à l'aide du test de présence et de la récursion :

```
let rec process await_immediate s =
  present s then () else run await_immediate s
```

De nombreuses constructions du langage peuvent ainsi s'encoder à partir d'un noyau plus réduit. Nous y reviendrons au moment de la définition formelle du langage dans le chapitre 4. La présence de ces constructions dérivées dans le langage permet à la fois d'alléger la syntaxe des programmes, mais aussi de proposer des implémentations plus efficaces de ces constructions.

Signaux valués On associe à un signal une valeur, ou plus précisément un flot de valeurs, c'est-à-dire une valeur par instant où le signal est présent. On peut émettre plusieurs valeurs par instant sur un signal : on parle de *multi-émission*. Les valeurs émises sont combinées à l'aide d'une fonction donnée au moment de la définition du signal pour obtenir la valeur du signal à cet instant. Par exemple, on peut facilement calculer la somme des étiquettes d'un arbre d'entiers :

exemples/ch2/sum.rml

```
let process sum t =
  signal s default 0 gather (+) in
  preorder (fun v -> emit s v) t
  ||
  await s(x) in print_int x
```

La valeur du signal `s` au premier instant est obtenue en itérant la fonction de combinaison, ici l'addition, sur les valeurs émises en commençant par la valeur par défaut `0`. La lecture du signal se fait avec la construction **`await`**, qui attend l'émission du signal et exécute son corps en liant `x` à la valeur du signal. Il est important de noter que l'affichage du résultat se fera au second instant. En effet, en REACTIVEML, on ne connait pas les émetteurs potentiels sur un signal. Si l'on veut connaître la valeur du signal à un instant donné, il faut être sûr qu'aucune émission sur ce signal n'aura lieu par la suite. Une façon simple de garantir cela est d'attendre la fin de l'instant, et donc de n'accéder à la valeur du signal qu'à l'instant suivant. Cela permet également de donner un sens au programme suivant, qui est rejeté en ESTEREL :

```
let process await_emit s =
  await s(x) in emit s (x + 1)
```

Si on pouvait lire instantanément la valeur d'un signal, la valeur de `s` serait définie récursivement puisqu'un signal n'a qu'une seule valeur par instant. On peut donc réagir immédiatement à la présence d'un signal avec **`await immediate`**, mais il faut attendre un instant pour lire sa valeur.

Enfin, l'opérateur **`last`** permet d'accéder à la dernière valeur émise sur le signal [2]. Cela permet par exemple de maintenir la valeur d'un signal, comme dans l'exemple suivant :

```
let process sustain_v s =
  loop
    emit s (last s);
    pause
  end
```

```
val sustain_v : ('a , 'a) event -> unit process
```

Un signal a le type `('a, 'b) event` où `'a` est le type des valeurs émises et `'b` le type de la valeur lue. Les types sont identiques dans le cas de `sustain_v` puisque l'on émet la valeur précédente du signal. La valeur par défaut du signal a le type `'a`, alors que sa fonction de combinaison a le type `'a -> 'b -> 'b`. Lorsque l'on omet la valeur par défaut et la fonction de combinaison d'un signal, c'est-à-dire qu'on écrit **`signal`** `s` **`in`** `e`, la valeur du signal `s` est la liste des valeurs émises. Sa valeur par défaut est donc la liste vide `[]` et sa fonction de combinaison est la concaténation `(::)`.

Structures de contrôle Nous avons déjà vu deux structures de contrôle en REACTIVEML : le test de présence et la boucle inconditionnelle. On dispose également des structures de contrôle

2. Il est noté `last?` en REACTIVEML classique.

habituelles en OCAML, comme les boucles **for** (dont une version parallèle) et **while** ou encore le **if**/**then**/**else**. Nous allons maintenant présenter deux structures propres aux langages synchrones : la préemption et la suspension.

La préemption utilise la construction **do** e **until** s **done**. Elle exécute son corps e jusqu'à l'émission du signal s. On parle de préemption faible, puisque le corps est exécuté quel que soit le statut du signal. C'est simplement à la fin de l'instant que l'on regarde la présence du signal et que l'on choisit d'interrompre le corps. Cette construction permet par exemple d'écrire un processus switch où chaque émission sur s_in modifie le statut de s_out. Ainsi, s_out est présent jusqu'à la première émission de s_in, puis est absent jusqu'à sa seconde émission, et ainsi de suite :

```
let process sustain s =
  loop emit s; pause end

let process switch s_in s_out =
  loop
    do run sustain s_out until s_in done;
    await immediate s_in; pause
  end
```

La construction **do**/**until** accepte aussi une continuation à exécuter lorsque le signal est émis et qui peut accéder à la valeur du signal. On peut utiliser cette fonctionnalité pour programmer un combinateur reset_every qui réinitialise un processus p à chaque nouvelle émission sur s :

```
let rec process reset_every s p v =
  do
    run p v
  until s(v) -> run reset_every s p v done
```

La suspension se fait avec la construction **do** e **when** s **done** qui exécute son corps e uniquement aux instants où le signal s est présent. Le corps reste bloqué si le signal est absent. Cela nous permet d'étendre l'exemple précédent pour obtenir un combinateur suspend_resume qui permet de suspendre et reprendre l'exécution d'un processus p à chaque émission du signal s :

```
let process suspend_resume s p =
  signal active in
  do run p when active done
  ||
  run switch s active
```

Aspects dynamiques Une des grandes particularités de REACTIVEML par rapport aux langages synchrones traditionnels comme ESTEREL est que l'on peut créer dynamiquement des processus. On utilise pour cela la combinaison de la récursion et du parallélisme synchrone. C'était déjà le cas dans notre premier exemple du processus levelorder (page 15) qui créé un processus par nœud de l'arbre. De façon similaire, on peut également programmer un map parallèle, qui lance en parallèle une instance du processus p par élément de la liste l :

```
let rec process par_map p l =
  match l with
  | [] -> []
  | x :: l -> let x' = run p x
              and l' = run par_map p l in
              x' :: l'
val par_map : ('a -> 'b process) -> 'a list -> 'b list process
```

La construction **let**/**and** exprime également le parallélisme synchrone, mais permet de récupérer les valeurs retournées par les deux branches. Un autre exemple de création dynamique est le combinateur foreach qui lance une instance du processus p à chaque émission du signal s :

```
let rec process foreach s p =
  await s(v) in
  run p v || run foreach s p
```

Une autre particularité du langage est que les processus et les signaux sont des objets de première classe. Ils peuvent donc être donnés en argument des fonctions, mais aussi stockés dans des structures de données ou encore émis sur un signal. Dans l'exemple suivant, un serveur de calcul reçoit sur le signal `add` un couple formé d'un processus `p` et d'un signal `ack` sur lequel il envoie le résultat de l'exécution de `p` :

```
let process server add =
  run foreach add (proc (p, ack) -> let v = run p in emit ack v)
```

Le mot-clé **proc** permet de définir un processus anonyme avec arguments. C'est l'équivalent du mot-clé **fun** pour les fonctions.

Un exemple complet : le problème des n-corps Pour terminer cette petite introduction à REACTIVEML, nous allons présenter un exemple complet : la simulation du problème des n-corps[3]. Le but est de simuler les interactions gravitationnelles de n corps, typiquement des planètes en orbite autour du soleil. Les lois du mouvement de Newton nous donnent le système d'équations différentielles suivant, où $\mathbf{x_i}$ désigne la position du ième corps (les vecteurs sont notés en gras) :

$$\forall i \in [1,n].\ m_i \ddot{\mathbf{x}}_\mathbf{i} = m_i \mathbf{a_i} = \sum_{j \neq i} \mathbf{f}_{ji}(\mathbf{x_i}) = -G \sum_{j \neq i} \frac{m_i m_j (\mathbf{x_i} - \mathbf{x_j})}{\mid \mathbf{x_i} - \mathbf{x_j} \mid^3}$$

Nous allons calculer une approximation de la solution à l'aide de méthodes numériques de résolution d'équations différentielles. Nous allons utiliser ici la méthode d'Euler semi-implicite[4][5]. Pour une équation de la forme $\ddot{x} = f(x)$ à partir d'une date de départ t_0, une position initiale x_0, une vitesse initiale v_0 et un pas de simulation dt, on calcule une suite de valeurs $\hat{x}_k \approx x(t_0 + k.dt)$ et $\hat{v}_k \approx \dot{x}(t_0 + k.dt)$, par :

$$\hat{v}_{k+1} = \hat{v}_k + dt.f(\hat{x}_k)$$
$$\hat{x}_{k+1} = \hat{x}_k + dt.\hat{v}_{k+1}$$

L'idée du programme REACTIVEML, visible sur la figure 2.1, est d'utiliser un processus par corps et un signal global `env` pour la communication. A chaque instant, chaque processus va émettre sur ce signal son attraction sous forme de champ de forces, c'est-à-dire une fonction associant une force à une position. La fonction de combinaison du signal fait ensuite la somme des champs de forces. Chaque corps lit ensuite le signal et utilise le résultat pour calculer sa position `dt` plus tard. Le programme principal lance en parallèle 100 planètes en utilisant une boucle **for** parallèle.

On peut très simplement ajouter des aspects plus dynamiques à cette simulation. On peut par exemple écrire un processus qui ajoute une nouvelle planète à chaque émission d'un signal `add`. On utilise à nouveau la combinaison de la récursion et du parallélisme synchrone :

@ exemples/nbody/planets_dyn.rml

```
let rec process add_planet add =
  await add(p) in
  run body p || run add_planet add
```

Maintenant que l'on a un exemple complet intéressant, on va chercher à exécuter ce programme. Pour cela, il faut d'abord compiler le fichier `planets.rml`, ce qui va générer un fichier OCAML `planets.ml`. On peut ensuite compiler ce fichier, sans oublier de le lier au moteur d'exécution de REACTIVEML, qui est une bibliothèque OCAML nommée `rmllib`. L'outil `rmlbuild`[6] permet de simplifier toutes ces étapes. Il suffit d'appeler :

3. http://fr.wikipedia.org/wiki/Problème_à_N_corps
4. http://fr.wikipedia.org/wiki/Méthode_d%27Euler_semi-implicite
5. La méthode d'Euler habituelle (ou explicite) ne peut pas être utilisée pour le problème des n-corps, puisqu'elle ne conserve pas l'énergie. On l'obtient en remplaçant $\hat{v}_{k+1}$ par $\hat{v}_k$ dans la seconde équation.
6. Cet outil est une extension du logiciel `ocamlbuild` de la distribution OCAML, développé au cours de cette thèse.

@ exemples/nbody/planets.rml

```
let dt = 0.01
signal env default (fun _ -> zero_vector) gather add_force

let rec process body (x_t, v_t, w) =
  emit env (force (x_t, w));
  await env(f) in
  let v_tp = v_t ++. (dt **. (f x_t)) in
  let x_tp = x_t ++. (dt **. v_tp) in
  draw_body x_t x_tp w;
  run body (x_tp, v_tp, w)

let process main =
  init_gui ();
  for i = 1 to 100 dopar
    run body (random_planet ())
  done
  ||
  run body (sun ())
```

FIGURE 2.1 – Simulation des n-corps

```
> cd exemples/nbody
> rmlbuild planets.rml.native
> ./planets.rml.native
```

Puisque le compilateur REACTIVEML génère du code OCAML, il est possible d'appeler n'importe quelle fonction OCAML depuis REACTIVEML. On peut en particulier accéder à tous les modules de la librairie standard OCAML. [7]

2.2 La programmation orientée agent en REACTIVEML

Le principe de la *programmation orientée agent* [Sho93] est de voir un programme comme l'interaction entre multiple *agents* (ou *acteurs*) autonomes, avec chacun leur propre état et « volonté ». C'est un cas particulier de programmation orientée objet. La communication entre les agents se fait par envoi de message non bloquant ou requête bloquante (qui attend une réponse). Il n'y a donc pas de mémoire partagée entre les agents. Ce paradigme de programmation est utilisé notamment par la bibliothèque JADE [BPR99] basée sur JAVA mais aussi par la bibliothèque SIMDIASCA basée sur ERLANG qui a inspiré ce travail. L'avantage de l'approche synchrone pour la programmation orientée agent est qu'elle permet de garantir simplement l'ordre des événements : on est sûr que tous les agents voient bien les causes avant les effets.

Un exemple simple : un capteur Le concept d'agent apparait très naturellement en REACTIVEML, sans même avoir à y penser. Il est par exemple utilisé dans les simulations de réseaux de capteurs écrites en REACTIVEML [SMMM06, MB05]. Un agent est simplement un processus qui communique avec les autres processus par envoi de messages à l'aide de signaux.

La figure 2.2 montre un exemple de nœud dans un réseau de capteur. Il reçoit des messages des autres capteurs sur le signal `me` (ligne 10), puis transmet le message à ses voisins après l'avoir décrémenté (`iter_p` itère un processus sur une liste, comme `List.iter`). Le reste du nœud modélise la consommation d'énergie. L'énergie du nœud, stockée dans le signal `energy`, est

7. Le typeur de REACTIVEML ne gère pas les modules, les objets ou encore les variants polymorphes. On ne peut donc pas y accéder directement.

@ exemples/sensor/radio.rml

```
let process node me neighbors =
  signal dead in
  signal energy default e_0 gather (fun x _ -> x) in
  let process send msg n = emit n msg in
  let process forward_msg msg =
    if msg>1 then run iter_p (send (msg-1)) neighbors
  in
  do
    loop (* protocol *)
      await me(msgs) in run iter_p forward_msg msgs
    end
    ||
    loop (* power *)
      if last energy < e_min
      then emit dead
      else emit energy (last energy -. max_power);
      pause
    end
  until dead done
```

FIGURE 2.2 – Un exemple d'agent modélisant un capteur

décrémentée de `max_power` à chaque pas de la simulation (ligne 16). Lorsque cette énergie descend sous `e_min`, le nœud est mort et cesse son exécution par une préemption sur le signal `dead`.

Vers une approche plus générique Nous allons maintenant tenter d'avoir une approche plus générique de la programmation orientée agent. Nous cherchons ici à illustrer l'expressivité du langage et à montrer un exemple typique de programme REACTIVEML qui nous servira de support dans la suite du manuscrit. Notre but est de créer une mini-bibliothèque pour faciliter les communications entre agents, la création et la terminaison des agents. Cela se fait très simplement puisque l'on dispose d'un langage d'ordre supérieur, dans lequel les signaux peuvent être utilisés comme des objets de première classe. Les fonctions de la bibliothèque appartiennent au module `Agents`. Nous allons illustrer le fonctionnement de la bibliothèque en reprenant l'exemple précédent :

@ exemples/sensor/radio_agent.rml

```
let process node self neighbors =
 signal energy default e_0 gather (fun x _ -> x) in
 (* reception des messages *)
 run Agents.react self
  begin proc msg _ ->
    if msg > 1 then
      iter (fun o -> Agents.send o (msg-1)) neighbors
  end
 ||
 (* comportement spontane *)
 loop
   if last energy < e_min
   then Agents.kill self
   else emit energy (last energy -. max_power);
   pause
 end
```

Le comportement d'un agent est décrit par un processus prenant en argument l'objet lui-même, appelé communément `self` dans la programmation orientée objet, et un paramètre supplémentaire

pour configurer l'objet, qui est ici la liste `neighbors` des voisins. Le corps du processus contient deux parties :

- La réponse aux messages des autres agents. On définit pour cela un processus (qui est ici anonyme) qui prend en entrée un nouveau message et le traite. On le donne ensuite en argument du processus `react` de la bibliothèque, qui lance une nouvelle instance de ce processus en parallèle à chaque réception de message. La fonction `send` permet d'envoyer un message à un autre agent.
- Le comportement autonome, « spontané » de l'agent. Il s'agit d'un processus lancé en parallèle de la réception des messages. Dans cet exemple, il s'agit de la boucle qui décrémente l'énergie du capteur. La terminaison du nœud se fait par un appel à la fonction `kill`. On utilise également un signal local pour représenter l'état interne du nœud.

Une mini-bibliothèque d'agents Regardons maintenant l'implémentation de cette mini-bibliothèque. Elle est suffisamment simple pour tenir sur une page, comme le montre la figure 2.3. Un agent est ainsi défini (ligne 7) par un identifiant `a_name`, un signal `a_msg_signal` pour la réception des messages des autres agents et un signal `a_kill_signal` permettant de terminer son exécution.

La création d'un agent se fait avec la fonction `mk_agent` (ligne 23), qui prend en argument le nom `name` de l'agent et son comportement `p`. Le comportement est un processus qui prend en argument le lien `self` vers l'objet et un paramètre supplémentaire `args` permettant de configurer l'agent. La fonction `mk_agent` crée la structure de données `self` représentant l'agent, puis le corps de l'agent qui lance le comportement `p` à l'intérieur d'une préemption permettant de le tuer.

Notre mini-bibliothèque d'agents permet d'envoyer des messages non bloquants et des requêtes bloquantes. Pour cela, un message est soit de la forme `Mmsg m`, soit une requête `Mreq(m, s)` où `m` est le message et `s` est le signal sur lequel la réponse sera envoyée. L'émission d'un message non bloquant se fait avec la fonction `send` (ligne 40) et correspond à l'émission sur le signal `a_msg_signal` de l'objet. On note aussi `o << m` pour `send o m` (ligne 42). Dans le cas d'une requête, implémentée par la fonction `send_and_wait` (ligne 45), on crée d'abord un signal local `s` que l'on envoie dans le message, puis on attend la réponse sur ce signal.

Pour la réception des messages, l'agent doit appeler le processus `react`, qui prend en argument l'agent `self` et un processus `act` qui traite un message entrant. A chaque réception de messages sur le signal `self.a_msg_signal`, le processus `react` lance une nouvelle instance du processus `act` pour chaque message reçu puis attend le message suivant. Pour gérer les différents types de messages, on utilise le processus `proxy`. Dans le cas d'une requête bloquante (ligne 17), il appelle le processus `act` avec une fonction envoyant le résultat sur le signal `s` de réponse contenu dans la requête. Dans le cas d'un message non bloquant (ligne 16), il appelle `act` avec une fonction qui ignore son argument. Le processus `act` ne doit donc appeler son dernier argument `return` que pour les requêtes bloquantes.

Messages et requêtes Pour illustrer l'utilisation des requêtes, nous allons écrire un second exemple que nous utiliserons dans la suite pour illustrer de nouvelles constructions. Il s'agit d'une banque, qui permet de déposer de l'argent et d'observer le solde de son compte :

exemples/bank/bank1.rml

```
type cmd = Put of int | Info

let process bank self () =
  signal counter default 0 gather (+) in
  run react self
    begin proc msg return -> match msg with
    | Put x -> emit counter x
    | Info -> return (last counter)
    end
  ||
  run sustain_v counter
```

exemples/sensor/agents.rml

```
(* Type des messages: message ou requete *)
type ('msg, 'out) msg =
  | Mmsg of 'msg
  | Mreq of 'msg * ('out, 'out list) event

(* Type des agents *)
type ('msg, 'out) agent = {
    a_name : string ;
    a_msg_signal : (('msg, 'out) msg, ('msg, 'out) msg list) event;
    a_kill_signal : (unit, unit list) event;
  }

let rec process react self act =
  (* Appelle le processus react avec le bon argument return *)
  let process proxy react msg = match msg with
  | Mmsg msg -> run react msg (fun _ -> ())
  | Mreq (msg, s) -> run react msg (fun v -> emit s v)
  in
  await self.a_msg_signal(msgs) in
  run iter_p (proxy act) msgs || run react self act

(* Creation d'un agent*)
let process mk_agent name p =
  signal kill, msg_sig in
  let self = { a_name = name; a_msg_signal = msg_sig;
               a_kill_signal = kill } in
  (* Processus principal d'un agent *)
  let process agent args =
    do
      run p self args
    until kill done
  in
  agent, self

(* Terminaison d'un agent *)
let kill o =
  emit o.a_kill_signal

(* Envoi d'un message *)
let send o msg =
  emit o.a_msg_signal (Mmsg msg)
let (<<) o msg = send o msg

(* Envoi d'une requete et attente du resultat *)
let process send_and_wait o msg =
  signal s in
  emit o.a_msg_signal (Mreq (msg, s));
  await immediate one s(v) in v
```

FIGURE 2.3 – La programmation générique d'agents

On définit tout d'abord le type algébrique des messages, qui est soit un dépôt `Put v`, soit une demande de solde `Info`. L'état de la banque est représenté par un signal `counter` donnant le solde actuel. Le comportement spontané de l'agent consiste à maintenir la valeur du solde. Pour ajouter de l'argent, on émet cette somme sur le signal `counter` (on remarque que la fonction de combinaison de ce signal est l'addition). Pour les demandes de solde, on répond à l'appelant en donnant la valeur précédente du solde à la fonction `return`.

Ainsi, si `b` est un agent simulant une banque, il faut appeler les commandes suivantes pour réaliser un dépôt de dix euros puis consulter le solde du compte à l'instant suivant :

```
b << (Put 10);
pause;
let v = run send_and_wait b Info in
print_endline ("Solde: "^string_of_int v)
```

2.3 Une première extension : les signaux à mémoire

Signal à mémoire Nous avons vu dans l'exemple précédent de la banque qu'il était nécessaire de maintenir manuellement la valeur du solde. Nous allons maintenant présenter notre première contribution qui est une extension de la notion de signal, appelée *signal à mémoire* ou mémoire, permettant d'automatiser cette tâche. Nous verrons dans la suite qu'il s'agit d'une extension légère à la fois du point de vue de la sémantique et de l'implémentation, qui ne requiert que de petites modifications. Cela permet de justifier cet ajout dans le langage, dont le rapport coût/bénéfice nous semble largement positif.

L'idée du signal à mémoire est simple : il s'agit d'un signal qui maintient sa valeur automatiquement. Lorsque l'on émet une valeur sur ce signal, le calcul de la valeur du signal se fait en appliquant la fonction de combinaison non pas en partant de la valeur par défaut, mais en partant de la valeur précédente du signal. L'exemple suivant montre la différence entre un signal « normal », déclaré par le mot-clé **`signal`**, et un signal à mémoire déclaré par le mot-clé **`memory`** :

exemples/ch2/signal_memory.rml

```
let process signal_memory =
  memory m default 0 gather (+) in
  signal s default 0 gather (+) in
  emit m 2; emit s 2; pause;
  emit m 1; emit s 1; pause;
  print_endline ("m="^string_of_int (last m));
  print_endline ("s="^string_of_int (last s))
> ./signal_memory.rml.native
m=3
s=1
```

L'exemple déclare un signal `s` et un signal à mémoire `m` avec les mêmes valeur par défaut et fonction de combinaison. On émet ensuite sur `s` et `m` la valeur `2` au premier instant, puis `1` au second instant. Enfin, on lit leur valeur précédente au troisième instant. Dans le cas de `s`, on lit `1 = 0+1`, puisque le calcul de la valeur est fait en partant de la valeur par défaut `0`. Dans le cas de `m` on lit `3 = (0+2)+1`, puisque l'on part de la valeur précédente `2`.

L'utilisation d'une mémoire nous permet de simplifier l'écriture de l'exemple précédent de la banque, puisqu'il n'est plus nécessaire de maintenir la valeur de `counter` :

exemples/bank/bank1_memory.rml

```
let process bank self () =
  memory counter default 0 gather (+) in
  run react self
    begin proc msg return -> match msg with
    | Put x -> emit counter x
    | Info -> return (last counter)
    end
```

Une fonction de combinaison générique Un des inconvénients de l'utilisation de signaux pour gérer un état partagé est que l'on doit connaître la fonction de combinaison du signal pour connaître l'effet de cette émission. Par exemple, dans le cas de la banque, on écrit :

```
emit counter x
```

alors que, si on avait été en OCaml et que l'on avait utilisé une référence, on aurait écrit :

```
counter := !counter + x
```

qui montre bien plus clairement le résultat de l'opération sur l'état partagé. On peut retrouver ce comportement en ReactiveML grâce à une fonction de combinaison astucieuse [8] :

```
memory s default d gather (fun f acc -> f acc) in ...
```

L'idée est de ne pas émettre une valeur sur un signal, mais une fonction qui prend en argument l'ancien état et renvoie le nouvel état. L'autre avantage de cette approche est qu'elle permet de faire des opérations différentes sur l'état partagé. On peut par exemple étendre notre exemple de banque pour ajouter des intérêts à la fin de chaque année. Pour cela, on émet tous les 12 instants une fonction qui ajoute 3% au solde :

exemples/bank/bank2.rml

```
let process bank self () =
  memory counter default 0 gather (fun f acc -> f acc) in
  run react self
    begin proc msg return -> match msg with
    | Put x -> emit counter ((+) x)
    | Info -> return (last counter)
    end
  ||
  loop
    for i=1 to 11 do pause done;
    emit counter (fun c -> int_of_float (1.03 *. (float_of_int c)));
    pause
  end
```

On peut aussi définir plusieurs notations utiles pour manipuler des signaux avec cette fonction de combinaison générique :

```
let (<<-) s f = emit s f
let (<==) s v = emit s (fun _ -> v)
```

La fonction `(<<-)` est une version infixe de **emit**. La fonction `(<==)` permet de donner une valeur précise à un signal à mémoire utilisant cette fonction de combinaison générique.

2.4 Vers les domaines réactifs

Après cet aparté présentant l'extension du langage avec la notion de signal à mémoire, nous allons revenir à notre sujet principal. Nous allons montrer les limites de ReactiveML et les problèmes que l'on cherche à résoudre par l'introduction des domaines réactifs.

Un problème de modularité Puisque l'on peut tester la présence d'un signal, il est clair qu'un processus qui répond instantanément n'est pas équivalent à un processus qui prend un instant pour répondre. Cela pose des problèmes en ReactiveML puisque certaines communications prennent du temps. Considérons par exemple le processus `sig_incr` qui attend son entrée sur le signal `s_in`, incrémente cette valeur de 2 puis émet le résultat sur le signal `s_out` :

```
let process sig_incr s_in s_out =
  loop await s_in(v) in emit s_out (v + 2) end
```

s_in	2	3	.	4	5	...
s_out	.	4	5	.	6	...

8. On appréciera au passage la beauté et la concision de l'approche fonctionnelle dans ce cas.

Puisqu'il faut un instant pour lire la valeur d'un signal, ce processus prend un instant pour calculer son résultat, c'est-à-dire que l'émission sur `s_out` se fait un instant après l'émission de l'entrée sur `s_in`. Supposons maintenant que l'on souhaite décomposer ce processus en deux parties qui vont chacune incrémenter de 1 leur entrée. On crée pour cela un signal local `tmp` qui va servir à relier les deux processus :

```
let process sig_incr_local s_in s_out =
  signal tmp default 0 gather (+) in
  loop await s_in(v) in emit tmp (v+1) end
  ||
  loop await tmp(v) in emit s_out (v+1) end
```

s_in	2	3	.	4	5	...
tmp	.	3	4	.	5	...
s_out	.	.	4	5	.	...

Il faut maintenant deux instants à ce processus pour calculer sa sortie. Il n'a donc pas le même comportement que le processus initial. Il existe en fait un moyen de contourner ce problème dans certains cas. Il faut pour cela utiliser la construction **`await immediate one`** qui permet d'obtenir immédiatement une des valeurs émises sur un signal. Son résultat est déterministe si une seule valeur est émise sur le signal par instant, comme c'est le cas pour le signal `tmp`. Sinon, elle renvoie une des valeurs émises, sans aucune garantie. Notre exemple corrigé devient donc :

```
let process sig_incr_refined s_in s_out =
  signal tmp in
  loop await s_in(v) in emit tmp (v+1) end
  ||
  loop
    await immediate one tmp(v) in
    emit s_out (v+1)
  end
```

s_in	2	3	.	4	5	...
tmp	.	3	4	.	5	...
s_out	.	4	5	.	6	...

Cette solution repose sur le fait que l'on est sûr qu'une seule valeur sera émise par instant. Elle ne fonctionne plus si plusieurs valeurs sont potentiellement émises. Par exemple, dans l'exemple suivant, on ne peut pas faire en sorte que `sig_filter` et `sig_filter_local` répondent à la même vitesse :

```
let process sig_filter s_in s_out =
  await s_in(v) in emit s_out ((v*v + (v - 15))*2)

let process sig_filter_local s_in s_out =
  signal tmp default 0 gather (+) in
  await s_in(v) in emit tmp (v*v)
  ||
  await s_in(v) in emit tmp (v - 15)
  ||
  await tmp(v) in emit s_out(v*2)
```

On voit donc que le raffinement est loin d'être évident en REACTIVEML. Les domaines réactifs vont permettre de résoudre ce problème en cachant des instants locaux.

État partagé Nous avons vu dans les exemples précédents comment utiliser un signal à mémoire pour représenter l'état d'un agent. Nous avons montré une banque un peu particulière, puisqu'elle ne permet que de déposer de l'argent, pas de le retirer. Une première approche pour ajouter cette fonctionnalité est de simplement ajouter un nouveau type de messages :

exemples/bank/bank3_wrong.rml

```
type cmd = ... | Take of int
type answer = Balance of int | Status of bool
```

```
let process bank self () =
  memory counter default 0 gather (fun f acc -> f acc) in
  run react self
    begin proc msg return -> match msg with
    | Put x -> emit counter (fun c -> c + x)
    | Info -> return (Balance (last counter))
    | Take x ->
        if x <= last counter then
          (emit counter (fun c -> c - x); return (Status true))
        else
          return (Status false)
    end
```

On doit définir le type `answer` des valeurs renvoyées par l'agent suite à des requêtes. On utilise ici un type algébrique puisqu'il s'agit soit d'un entier pour le solde, soit d'un booléen pour le résultat d'un retrait. Pour le retrait, on vérifie si le solde est suffisant, puis on le décrémente et on répond à l'émetteur.

Malheureusement, cette version simple n'a pas du tout le comportement attendu. Si la banque reçoit au même instant deux requêtes de retrait inférieures au solde, mais dont la somme est supérieure au solde, alors on se retrouvera avec un solde négatif à l'instant suivant, ce que l'on veut éviter. En effet, le traitement de chacune des requêtes ne regarde que la valeur du solde à l'instant précédent et ne prend pas en compte les autres requêtes simultanées. Il faut donc centraliser les opérations de retrait. Pour cela, on ajoute un signal local `req` sur lequel sont envoyées les demandes de retrait et un processus qui écoute ces demandes et les traite séquentiellement :

@ exemples/bank/bank3_fixed.rml

```
let process bank self () =
  ...
  run react self
    begin proc msg return -> match msg with
    ...
    | Take x -> emit req (x, return)
    end
  ||
  let try_take c (x,return) =
    if x <= c then (return (Status true); c - x)
    else (return (Status false); c)
  in
  loop
    await req(l) in
    let new_c = fold_left try_take (last counter) l in
    emit counter (fun c -> c - (last counter - new_c))
  end
```

La fonction `try_take` traite une requête, formée d'une somme `x` et d'une fonction de retour `return`, et renvoie le nouveau solde. On l'applique séquentiellement sur toutes les requêtes reçues sur le signal `req` avant de mettre à jour l'état partagé.

Cette version corrigée a bien le comportement attendu : le solde ne devient jamais négatif. Mais le soucis de cette approche est qu'il faut maintenant deux instants pour traiter les messages reçus par la banque. On perd donc la correspondance entre un pas de la simulation et un instant du programme. Nous verrons dans le paragraphe suivant à quel point cela complique l'écriture du programme. Cela ne posera plus de soucis avec les domaines réactifs puisque l'on pourra masquer les instants locaux.

Au vu de la difficulté que l'on a à écrire un agent aussi simple que la banque, on peut se demander comment on peut arriver à programmer des exemples plus conséquents. La réponse à cette question est que les programmeurs ReactiveML n'utilisent pas des signaux pour gérer un état

partagé que l'on souhaite modifier plusieurs fois par instant. A la place, ils utilisent des références comme dans cette version de la banque :

exemples/bank/bank3_ref.rml

```
let process bank self () =
  let counter = ref 0 in
  run react self
    begin proc msg return -> match msg with
    | Put x -> counter := !counter + x
    | Info -> return (Balance !counter)
    | Take x ->
        if x <= !counter then
          (counter := !counter - x; return (Status true))
        else
          return (Status false)
    end
```

L'implémentation séquentielle de REACTIVEML garantit l'atomicité de l'exécution de chaque expression ML et donc de chaque modification de la la référence partagée `counter`. Si les références permettent de résoudre le problème, alors pourquoi se pose-t-on toutes ces questions et pourquoi chercher à les remplacer ? On peut apporter deux réponses :

- Cela ne correspond pas à la sémantique formelle : l'esprit du langage est de communiquer en utilisant des signaux avec diffusion instantanée. Recourir à des effets de bords dans ce contexte devrait donc être interdit.
- Les références ne sont pas utilisables dans une implémentation parallèle ou distribuée. Dans le cas distribué, la notion de mémoire partagée disparait complètement. Dans le cas parallèle, en mémoire partagée, on perd les propriétés d'atomicité de l'accès aux références, ce qui rend leur utilisation hasardeuse.

Instants logiques et pas de simulation Lorsque l'on programme une simulation en REACTIVEML, il est très naturel de faire correspondre chaque instant du programme à un pas de temps de la simulation. Malheureusement, les exemples précédents nous ont montré qu'il est parfois impossible de simuler un pas de temps d'un agent en un seul instant, sauf à sacrifier drastiquement la modularité et la lisibilité du programme. Nous allons maintenant montrer comment maintenir la synchronisation des agents dans le cas où il faut plusieurs instants pour simuler chaque pas de temps d'un agent.

La première chose à faire est de distinguer les différents pas de temps de la simulation. Un nouveau pas de temps commence dès que tous les agents ont fini le pas de temps précédent. Nous allons maintenant programmer un petit ordonnanceur chargé de gérer ce passage des pas de temps. Il est constitué de :

- Un signal `step` indiquant le début d'un pas de temps.
- Un signal `move` émis à chaque instant par les agents actifs.
- L'ordonnanceur lui-même qui émet le signal `step`, puis attend que le signal `move` soit absent, et ainsi de suite :

```
let rec process await_absent s =
  present s then (pause; run await_absent s) else ()

let process scheduler =
  loop
    emit step;
    run await_absent move
  end
```

Tout cela semble simple et ne pas demander d'efforts particuliers. Pourtant, il faut maintenant être très attentif à la façon de programmer les agents. Par exemple, si on veut attendre le prochain

pas de la simulation, il ne suffit plus d'utiliser l'opérateur **pause**. Il faut maintenant appeler le processus suivant :

```
let process pause_ =
  pause;
  await immediate step
```

De la même façon, la gestion des signaux est rendue bien plus complexe. Ainsi, il faut par exemple maintenir la valeur des signaux permettant la communication avec les autres agents pendant toute la durée d'un pas de simulation. On souhaite en effet comme dans notre mini-bibliothèque que les agents ne communiquent qu'une fois par pas de temps. Il faut donc accumuler les messages envoyés par les autres agents au cours d'un pas de simulation, qui correspond à plusieurs instants du programme. Cela peut paraitre simple en apparence, mais il faut faire attention à ne maintenir le signal qu'entre deux occurrences de `step`. Pour obtenir ce résultat, on ajoute un nouveau signal `eoi` qui sert à terminer le maintien des signaux, qui est géré par le processus `hold_next_step`.

```
let process scheduler =
  loop
    emit step;
    run await_absent move;
    emit eoi; pause; pause
  end

let process hold_next_step s =
  loop
    await immediate step; pause;
    do
      run sustain_v s
    until eoi done;
  end
```

Le processus `hold_next_step` attend le début du pas puis maintient le signal jusqu'à la fin du pas, c'est-à-dire l'émission du signal `eoi`. Il faut également utiliser le processus `await_` pour attendre un signal :

```
let process await_ s =
  await s(_) in
  await immediate eoi;
  let v = last s in
  await immediate step;
  v
```

Il faut veiller à ne lire la valeur du signal qu'après la fin du pas de simulation, après l'émission de `eoi`, et pas au moment de la première émission du signal.

La programmation de ces différents processus est très complexe et source d'erreurs. Il n'est par ailleurs pas très évident de savoir pourquoi on appelle deux fois **pause** dans le processus `scheduler`, ou encore pourquoi on appelle **pause** dans le processus `hold_next_step` mais pas dans `await_`. En outre, on ne programme plus vraiment en ReactiveML, mais dans une sorte de langage embarqué dans ReactiveML. On perd ainsi de nombreuses propriétés du langage, qui reposent maintenant sur le bon vouloir du programmeur d'utiliser les constructions du langage embarqué de la façon attendue. L'autre remarque importante est que l'on est en quelque sorte en train d'implémenter un interprète ReactiveML en ReactiveML. En effet, le rôle du moteur d'exécution est déjà de décider de la fin de l'instant, lorsque tous les processus ont terminé, et de gérer les signaux, notamment en lançant au bon moment les processus en attente d'un signal. La notion de domaines réactifs permet de laisser le moteur d'exécution gérer les signaux et la synchronisation des processus, au lieu de devoir les programmer à la main. Les quelques processus que l'on vient de voir donnent une approche informelle de la sémantique des domaines réactifs. L'annexe A montre une approche plus

systématique d'encodage des domaines réactifs en REACTIVEML classique, qui reprend la plupart des idées présentées ici.

Conclusion Lorsque l'on écrit une simulation discrète, par exemple de réseau de capteurs, il est naturel de structurer la simulation en pas de temps pendant lesquels chacun des agents exécute une action et communique une fois avec les autres agents. Ce pas de temps correspond le plus souvent à une durée fixe ou variable dans le temps de la simulation. Le choix le plus simple pour écrire une telle simulation en REACTIVEML est que chaque instant logique corresponde à un pas de la simulation. Ainsi, en connaissant le lien entre le pas de temps et le temps de la simulation, on peut par exemple attendre une certaine durée en appelant **pause** autant que nécessaire. De même, les signaux permettent naturellement de communiquer une fois par pas de temps.

Malheureusement, il est souvent difficile de programmer un pas de temps d'un agent en un seul instant logique. Nous avons vu par exemple que la gestion d'un état partagé sans références requiert plusieurs instants logiques. De même, la décomposition d'un agent en plusieurs agents communiquant entre eux peut aussi casser cet invariant, puisque certaines communications prennent du temps.

Les domaines réactifs vont résoudre ce problème. Ils permettront d'utiliser un nombre d'instants quelconque pour modéliser chaque pas de temps d'un agent, puis de masquer ces instants locaux. Le lien entre pas de temps et instant logique sera donc conservé et simplifiera l'écriture des programmes. En outre, la distinction entre synchronisations locales et synchronisations globales permettra de paralléliser ou distribuer le code de façon simple.

CHAPITRE 3

Une présentation informelle des domaines réactifs

Nous allons présenter dans ce chapitre la notion de *domaine réactif*, qui est la principale contribution de cette thèse. Nous montrerons plusieurs exemples typiques de programmation avec les domaines et nous expliquerons ce qui rend nécessaires les analyses statiques que nous présenterons dans la suite du document. Nous montrerons ensuite plusieurs exemples plus conséquents, en revenant notamment sur la programmation orientée agent.

3.1 Les domaines réactifs

Considérations « philosophiques »

Avant de rentrer dans la description proprement dite des domaines réactifs, nous allons tenter de donner une intuition de ce concept. Nous avons vu à la fin du chapitre précédent les motivations pragmatiques de cette extension du langage. Nous allons ici discuter de motivations plus « philosophiques ».

Le modèle synchrone, comme beaucoup d'autres modèles de concurrence, est à deux étages : on a d'un côté les processus qui se déroulent dans le temps, sur une échelle de temps commune, et de l'autre les fonctions instantanées qui permettent d'effectuer des calculs au sein des instants. Ces dernières sont le plus souvent programmées dans un langage généraliste, sans concurrence. En REACTIVEML, cette séparation entre fonctions et processus est syntaxique. Elle est aussi particulièrement visible dans les langages comme LUSTRE et ESTEREL v5 dont l'expressivité est limitée et qui recourent à l'import de fonctions externes, écrites par exemple en C.

Pourtant, les langages synchrones réactifs issus de REACTIVEC, notamment REACTIVEML, ont montré que le modèle de concurrence synchrone pouvait être utile dans un cadre plus général de programmation concurrente, sans aucune considération de temps-réel ni même de réactivité. Un de nos premiers exemples a montré que l'on pouvait exprimer le parcours en largeur d'un arbre très simplement grâce au parallélisme synchrone (le processus `levelorder` (page 15)). Nous allons donc chercher à pousser l'idée du synchrone jusqu'au bout : on veut pouvoir utiliser le parallélisme synchrone pour programmer n'importe quelle partie du programme. Bien entendu, il ne s'agit pas de programmer *toutes* les fonctions du programme en utilisant des processus et des signaux, mais plutôt d'en avoir la possibilité lorsque cela facilite l'écriture du programme.

Le concept de domaine réactif concrétise cette vision, en permettant en quelque sorte de lancer un programme REACTIVEML au sein d'un autre programme REACTIVEML. On dira ainsi que les domaines réactifs *réifient* le moteur d'exécution de REACTIVEML. La réification [FW84] est l'opération qui transforme un concept en objet concret. Par exemple, la fonction `eval` disponible dans de nombreux langages réifie l'interprète de ce langage en le rendant accessible dans un programme. Dans les langages fonctionnels, les lambda-expressions réifient la notion de fonction. Dans le cas des domaines réactifs, tout se passe en effet comme si l'on prenait le programme à

l'intérieur du domaine et qu'on le compilait. On obtient alors un programme séquentiel dont le point d'entrée est une fonction (que l'on appelle `react` dans le cas de REACTIVEML [1]) permettant d'exécuter un instant de ce programme. Le domaine réactif se comporte alors comme une boucle appelant cette fonction `react` un certain nombre de fois. Par exemple, il peut l'appeler jusqu'à ce que le programme termine et renvoie sa valeur. L'intégration du concept de domaine réactif dans le langage permet également de donner un sens à la communication entre ces deux programmes. La définition de la sémantique des domaines réactifs revient à faire les différents choix et restrictions qui assurent que cette communication a une sémantique bien définie.

Domaines et horloges

Un domaine réactif crée une notion d'instant local en subdivisant un instant en plusieurs instants locaux, inobservables de l'extérieur. On crée un domaine avec la construction :

```
domain ck do e done
```

où `ck` est l'identifiant du domaine, que l'on appellera son *horloge* et qui est lié dans le corps `e` du domaine. L'horloge représente l'échelle de temps attachée au domaine. C'est pourquoi l'opérateur **pause** prend maintenant en argument une horloge : **pause** `ck` attend le prochain instant du domaine d'horloge `ck`. On peut omettre l'argument de **pause**, ce qui correspond alors à attendre le prochain instant de l'horloge locale. On peut accéder à cette horloge locale avec la construction `local_ck` et on définit donc **pause** comme **pause** `local_ck`. L'exemple suivant affiche `"Hello"` pendant le premier instant du domaine d'horloge `ck` et `" world"` pendant le second :

exemples/ch3/hello_world_ck.rml

```
let process hello_world_ck =
  domain ck do
    print_string "Hello"; pause ck;
    print_endline " world"
  done
```

Les deux instants de l'horloge `ck` [2] sont inclus dans le premier instant de l'horloge globale du programme, celle d'un programme REACTIVEML classique, que l'on note `global_ck`. Le domaine termine instantanément lorsque son corps termine et renvoie la valeur de son corps. Les instants du domaine réactif sont inobservables de l'extérieur. Le processus ci-dessus est donc équivalent au processus suivant, dans lequel on a enlevé le domaine réactif et toutes les synchronisations sur l'horloge locale :

exemples/ch3/hello_world_seq.rml

```
let process hello_world_seq =
  print_string "Hello";
  print_endline " world"
```

L'imbrication des horloges au cours de l'exécution d'un programme définit un arbre étiqueté par les horloges, que l'on appelle *l'arbre d'horloges*. Un domaine réactif est le fils d'un autre s'il est défini dans la portée de celui-ci. Ainsi, l'arbre d'horloges correspondant à l'exemple précédent est constitué d'un nœud étiqueté par `global_ck`, qui est la racine de tous les arbres d'horloges, avec un seul fils étiqueté par `ck`. On appelle horloge parente d'une horloge `ck` l'horloge associée au père du nœud étiqueté par `ck` dans l'arbre d'horloges. On dira également qu'une horloge `ck1` est plus rapide qu'une horloge `ck2` si le nœud correspondant à `ck1` dans l'arbre d'horloges est un descendant de celui associé à `ck2`. Ainsi, `global_ck` est toujours l'horloge la plus lente du programme.

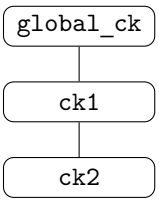
Notre premier exemple de domaine réactif terminait son exécution instantanément dans le premier instant de l'horloge globale. L'exécution d'un domaine réactif peut se dérouler sur plusieurs instants de son horloge parente. Pour cela, on peut créer un domaine réactif périodique avec le mot-clé **by**, comme dans l'exemple suivant :

1. Elle est définie dans le fichier @ `interpreter/rml_machine.ml`.
2. Dans la suite, on confondra souvent un domaine réactif et son horloge.

exemples/ch3/stutter.rml
```
let process stutter msg =
  domain ck by 6 do
    loop print_string msg; pause ck end
  done
```
Le processus `stutter` affiche six fois la chaine `msg` par instant de l'horloge globale, puisque le domaine d'horloge `ck` effectue six instants locaux par instant de son horloge parente. On peut écrire une expression quelconque après **by**, qui est évaluée à la création du domaine. Les instants de domaines réactifs frères (c'est-à-dire dont les horloges ont le même père dans l'arbre d'horloges) sont indépendants. Ils peuvent donc s'exécuter dans un ordre quelconque. Par exemple, `run stutter "a" || run stutter "b"` va afficher six caractères `'a'` et six caractères `'b'` par instant, mais dans un ordre quelconque.

On peut imbriquer librement les domaines réactifs. Ainsi, on peut réécrire le processus `stutter` de façon équivalente par :

```
let process stutter_nested msg =
  domain ck1 by 3 do
    domain ck2 by 2 do
      loop print_string msg; pause ck2 end
    done
  done
```



On peut voir à droite du programme l'arbre d'horloges correspondant si on lance ce processus dans l'horloge globale : à chaque instant de `global_ck`, on exécute trois instants de `ck1`, pour lesquels on exécute à chaque fois deux instants de `ck2`. On affiche donc bien le message six fois par instant de l'horloge globale.

Une horloge est une valeur de première classe que l'on peut, par exemple, donner en argument d'une fonction ou d'un processus. On peut ainsi réécrire l'exemple `stutter` comme :

exemples/ch3/stutter_clock.rml
```
let process print_loop msg pck =
  loop print_string msg; pause pck end

let process stutter msg =
  domain ck by 6 do run print_loop msg ck done
```

Domaines et signaux

La première modification du langage proposée est que **pause** prend maintenant en argument une horloge. La seconde est que chaque signal est désormais attaché au domaine réactif dans lequel il est défini : il a une valeur par instant de ce domaine. On dit aussi qu'un signal a l'horloge `ck` s'il est attaché au domaine réactif d'horloge `ck`. C'est pour cela que l'on a qualifié d'*horloge* l'identifiant d'un domaine réactif, en référence aux horloges dans les langages synchrones flot de données [Cas92]. Dans la suite, on appellera signal lent un signal d'horloge plus lente que l'horloge locale et signal rapide un signal dont l'horloge est l'horloge locale.

La sémantique d'un signal défini dans un domaine réactif est la même que celle des signaux habituels. Par exemple, on peut reprendre l'exemple du processus `sum` (page 17) et le lancer à l'intérieur d'un domaine réactif :

exemples/ch3/sum_ck.rml
```
let process sum_ck t =
  domain ck by 2 do
    signal s default 0 gather (+) in
    preorder (fun v -> emit s v) t
    ||
    await s(v) in print_int v
  done
```
Ce processus affiche la somme des éléments de l'arbre `t` au second instant de l'horloge locale `ck`, qui est inclus dans le premier instant de l'horloge globale du programme.

D'autre part, l'émission sur un signal lent ne pose aucun problème grâce à la multi-émission. La valeur du signal est obtenue en combinant toutes les valeurs émises au cours d'un instant de son horloge, ce qui comprend les valeurs émises au cours de tous les instants des horloges plus rapides inclus dans cet instant. Par exemple, le processus suivant va afficher 3 ($= 0 + 1 + 2$) :

exemples/ch3/emit_slow.rml
```
let process emit_slow =
  signal s default 0 gather (+) in
  domain ck by 2 do
    emit s 1; pause ck; emit s 2
  done
  ||
  await s(v) in print_int v
```

On peut aussi attendre l'émission d'un signal lent et récupérer sa valeur. Comme en REACTIVEML, il faut attendre l'instant suivant pour lire la valeur. Il faut prendre garde au fait qu'il s'agit de l'instant suivant de l'horloge du signal, comme le montre l'exemple suivant :

exemples/ch3/slow_signal.rml
```
let process slow_signal =
  signal s default 0 gather (+) in
  domain ck by 3 do
    await s(v) in print_int v
  done
  ||
  emit s 4
```

L'affichage du résultat (c'est-à-dire 4) a lieu dans le quatrième instant de l'horloge `ck`, qui est inclus dans le deuxième instant de l'horloge globale.

On peut également déclarer un signal lent en donnant son horloge avec le mot-clé **clock**. On peut ainsi écrire l'exemple précédent de façon équivalente comme :

exemples/ch3/slow_signal_clock.rml
```
let process slow_signal_clock =
  let pck = local_ck in
  domain ck by 3 do
    signal s clock pck default 0 gather (+) in
    await s(v) in print_int v
    ||
    emit s 4
  done
```

Nous verrons dans la suite plusieurs cas où cette possibilité est indispensable pour l'écriture des programmes. On peut omettre l'horloge du signal qui est alors égale à l'horloge locale `local_ck`.

La dernière nouveauté concernant les signaux est la *réinitialisation* d'un signal à chaque nouvel instant d'une horloge, que l'on déclare avec le mot-clé **reset** à la définition du signal. La dernière valeur du signal, accessible avec l'opérateur **last**, est réinitialisée à sa valeur initiale, c'est-à-dire la valeur par défaut du signal, au début de chaque instant de l'horloge en question. Cette réinitialisation est particulièrement utile dans le cas d'un signal à mémoire, puisque le calcul de la valeur se fait en partant justement de cette dernière valeur. Nous verrons dans la suite plusieurs exemples où la réinitialisation d'un signal à chaque instant d'une horloge apparait comme naturelle pour le programmeur. Nous allons l'illustrer avec l'exemple suivant :

exemples/ch3/memory_reset.rml
```
let process memory_reset =
  domain ck do
    signal s default 0 gather (+) reset global_ck in
    emit s 2;
    pause ck; print_int (last s);
    pause global_ck; print_int (last s)
  done
```

On émet d'abord `2` sur le signal `s` au cours du premier instant de `ck`. Au second instant local, on affiche donc `2`. On passe ensuite au second instant de l'horloge globale `global_ck`, ce qui signifie que la dernière valeur du signal est remise à sa valeur initiale. On affiche donc ensuite `0`.

Intuition du calcul d'horloges

En REACTIVEML, tous les processus peuvent accéder à un signal quel que soit l'endroit où il est défini. Ce n'est plus le cas lorsque l'on introduit plusieurs échelles de temps avec des domaines réactifs. Puisque le domaine masque ses instants locaux, on comprend bien que l'on ne peut pas utiliser un signal attaché à un domaine en dehors de ce domaine, puisque l'on ne voit pas les variations de sa valeur. En particulier, on ne peut donc pas retourner une valeur contenant un signal local, comme dans l'exemple suivant :

exemples/ch3/result_escape.rml

```
let process result_escape =
  domain ck by 10 do
    signal s in s
  done
```

Nous verrons plus tard qu'un programme qui accède à un signal en dehors de son domaine de définition n'a pas de sémantique. Nous définirons ensuite dans le chapitre 6 un système de types permettant de rejeter ce genre de programmes, que nous appellerons *calcul d'horloges*. Son principe est assez simple : on associe à chaque signal son horloge, puis on vérifie que l'horloge du domaine n'apparait pas dans le type du corps du domaine. Dans l'exemple suivant, le compilateur affichera le message d'erreur suivant :

```
The clock of this expression, that is,
('_a list, '_a) event{?ck|} process {'_c0|0}
depends on ck which escapes its scope.
```

Nous expliquerons la signification des types affichés dans ce message au moment de la présentation du calcul d'horloges.

Il faut également prendre en compte le fait que les signaux sont des objets de première classe dans le langage. On peut donc les manipuler comme n'importe quelle valeur et en particulier les émettre sur d'autre signaux. Cela permet aux signaux d'échapper de leur portée lexicale. Il faut donc être capable de rejeter les programmes où un signal échappe de son domaine en étant émis sur un signal lent, comme l'exemple suivant :

exemples/ch3/signal_escape.rml

```
let process signal_escape =
  signal slow in
  domain ck by 10 do
    signal fast default 0 gather (+) in
    emit slow fast
  done
```

```
The emitted value has clock (int , int) event{ck|},
and would thus escape its scope ck.
```

Le calcul d'horloges rejette ce programme car l'horloge locale `ck` apparait dans le type d'une variable définie en dehors du domaine, en l'occurrence `slow`. Mais parfois l'accès à un signal n'apparait pas dans le type d'une expression, par exemple si l'on retourne une fonction qui émet sur un signal :

exemples/ch3/effect_escape.rml

```
let process effect_escape =
  domain ck by 10 do
    signal fast in
    let f () = emit fast in f
  done
```

La solution classique à ce problème est d'utiliser la notion d'*effet* [LG88]. Le principe est d'associer à chaque expression non seulement un type, correspondant au résultat de l'exécution, mais aussi un effet représentant l'effet de l'exécution de cette expression. Dans notre cas, l'effet d'une expression est l'ensemble des horloges des signaux auxquels elle accède. Le compilateur va donc rejeter ce programme avec le message suivant :

```
The clock of this expression, that is, (unit =>{?ck} unit) process {'_c0|0}
depends on ck which escapes its scope.
```

Il existe une seconde restriction sur l'utilisation de signaux en présence de domaines réactifs, qui est moins évidente à comprendre. Il s'agit de l'interdiction de dépendre instantanément d'un signal lent. L'exemple suivant montre la source du problème :

exemples/ch3/immediate_dep_wrong.rml

```
let process immediate_dep_wrong =
  signal s in
  domain ck by 2 do
    await immediate s; print_string "Ok"
    ||
    pause ck; emit s
  done
```

```
This immediate dependency would make 'ck' escape its scope.
```

Pendant le premier instant de l'horloge `ck`, on considère le signal lent `s` comme absent. La première branche du parallèle est donc en attente, alors que la seconde attend l'instant suivant de `ck`. Au second instant de `ck`, le signal `s` est émis et devient donc présent. On devrait alors réveiller la première branche et afficher le message `"Ok"`. Nous voulons rejeter ce programme car il fait deux hypothèses différentes sur la présence du signal `s` au cours du même instant, ce qui est contraire au principe du synchrone.

On pourrait alors se dire que ce programme devrait afficher le message `"Ok"` au cours du premier instant de l'horloge `ck`, puisque le signal `s` est bien émis au cours de l'instant (de l'horloge globale). On respecte alors bien l'hypothèse du statut unique des signaux, mais on obtient un programme qui ne suit pas l'intuition de cause qui précède les effets. En effet, on a spéculé la présence de `s` au premier instant, alors que l'émission de `s` n'a lieu qu'au second instant de `ck`. L'information a donc en quelque sorte remonté le temps. On reconnait ici les notions de *causalité logique* et *causalité constructive* d'ESTEREL [Ber96] : ce programme est logiquement correct, mais pas constructivement correct. Nous reviendrons plus en détail sur le lien avec la causalité d'ESTEREL dans le chapitre 10.

Nous choisissons ici de ne pas donner de sémantique à un programme où l'on dépend instantanément d'un signal lent, comme dans l'exemple précédent. On veut en effet que tous les programmes soient causaux « par construction », comme en REACTIVEML classique. Les opérations imposant une dépendance immédiate, comme `await immediate` ou le test de présence, sont donc uniquement possibles si l'horloge du signal est égale à l'horloge locale. Là encore, le système de types rejettera les programmes (possiblement) incorrects en vérifiant cette condition.

Attente automatique des domaines réactifs

Considérons le domaine réactif suivant :

```
let process delayed_hello_world =
  signal s default "" gather (^) in
  domain ck by 10 do
    pause global_ck; emit s "Hello world"
    ||
    await s(v) in print_string v
  done
```

A la fin du premier instant de l'horloge `ck`, la première branche du parallèle est bloquée, en attente du prochain instant de l'horloge globale `global_ck`. La seconde branche est également en attente

d'un signal d'horloge `global_ck`, qui ne peut pas varier avant le prochain instant de l'horloge globale. Le corps du domaine n'évolue donc pas au cours du deuxième instant du domaine réactif, ni des huit instants suivants. On peut se dire qu'il n'est pas nécessaire d'exécuter dix instants de l'horloge locale `ck` : on peut détecter à la fin du premier instant que tous les processus à l'intérieur du domaine attendent le prochain instant de l'horloge globale et directement attendre ce prochain instant, sans avoir à exécuter les autres instants locaux dont on sait qu'ils sont inutiles.

On pourrait en rester là et simplement remarquer cette possibilité d'optimisation et l'utiliser au moment de l'exécution. Nous choisissons ici d'aller plus loin et d'intégrer cette notion à la sémantique des domaines réactifs. Ainsi, un domaine réactif décidera non seulement de la fin de son instant local, mais aussi d'attendre l'instant suivant de son horloge parente lorsque tous les processus qu'il contient sont bloqués en attente d'une horloge plus lente. Le nombre indiqué après le **by** ne représente donc pas le nombre d'instants locaux par instant de l'horloge parente, mais le nombre *maximal* d'instants que le domaine peut exécuter par instant de son horloge parente.

Ce choix permet de donner un sens aux programmes où l'on omet la borne du domaine. Dans ce cas, le domaine réactif effectue un nombre d'instants potentiellement non borné avant d'attendre le prochain instant de son horloge parente. Cela permet de définir un processus qui attend un arbre sur un signal `s`, puis effectue son parcours en largeur instantanément :

exemples/ch3/levelorder_ck.rml

```
let rec process levelorder_ck f s =
  domain ck do
    loop
      await s(t) in run levelorder f t;
      pause global_ck
    end
  done
```

Le processus `levelorder` est celui que l'on a défini au début du chapitre 2 (page 15). Le nombre d'instants effectués par le domaine d'horloge `ck` est égal à la profondeur de l'arbre `t`. Il est donc non borné et peut varier à chaque instant de l'horloge globale.

Une autre particularité de cette propriété d'attente automatique des domaines réactifs est qu'elle prend en compte la présence des signaux. Ainsi, un processus en attente d'un signal rapide non présent sera considéré comme bloqué et attendra l'instant suivant. En effet, si tous les autres processus à l'intérieur du domaine réactif sont également en attente du prochain instant d'une horloge plus lente, alors on est sûr que le signal rapide ne peut pas être émis avant le prochain instant de l'horloge lente. On peut donc attendre directement cet instant :

exemples/ch3/auto_waiting.rml

```
let process auto_waiting =
  domain ck do
    signal s in
    await immediate s; print_endline "Received"
    ||
    pause global_ck; emit s
  done
```

Le domaine réactif d'horloge `ck` n'exécute qu'un seul instant au cours du premier instant de l'horloge globale. Le message s'affiche au cours du second instant de `ck`, qui est inclus dans le second instant de l'horloge globale.

Domaines non coopératifs

On a vu que l'on peut omettre la borne sur le nombre d'instants du domaine réactif si l'on sait que tous les processus contenus dans le domaine vont à un certain point attendre le prochain instant d'une horloge plus lente. Mais que se passe-t-il si ce n'est pas le cas ? Le domaine réactif suivant n'attend jamais une horloge plus lente :

exemples/ch3/nonreactive_domain.rml
```
let process nonreactive_domain =
  domain ck do
    loop pause ck end
  done
```

Le domaine réactif va alors effectuer un nombre infini d'instants et se comporter comme une boucle infinie. Le programme complet ne pourra donc jamais passer à l'instant suivant de l'horloge globale. On dit alors que le domaine n'est pas *coopératif*, puisqu'il ne donne jamais la possibilité aux autres processus de passer à l'instant suivant. On dit également que ce programme n'est pas *réactif* puisqu'il ne répond plus à ses entrées.

Ce problème de non-réactivité des programmes n'est en fait pas nouveau. Il est déjà tout à fait possible d'écrire un programme REACTIVEML classique dans lequel les instants logiques ne progressent pas. C'est bien sûr le cas si l'on appelle une fonction récursive qui ne termine pas, comme dans n'importe quel programme OCAML, mais ce n'est pas le problème qui nous intéresse. Nous nous intéressons au cas où l'on définit un processus qui effectue une boucle instantanée au lieu de s'exécuter sur plusieurs instants. C'est le cas de l'exemple suivant, où l'on souhaite écrire un processus `timer` qui émet le signal `s` lorsque `delay` secondes se sont écoulées :

exemples/ch3/timer.rml
```
let process timer delay s =
  let time = ref (Unix.gettimeofday ()) in
  loop
    let time' = Unix.gettimeofday () in
    if time' -. !time >= delay
    then (emit s (); time := time')
  end
```

Pour implémenter ce processus, on initialise une référence `time` avec l'heure actuelle, obtenue en appelant la fonction `gettimeofday` du module `Unix` de la librairie standard. Ensuite, on boucle tant que le temps écoulé est inférieure à `delay`. Lorsqu'il devient supérieur à `delay`, on émet `s` et on met à jour la référence. Ce processus n'a pas le comportement attendu. En effet, le corps de la boucle est instantané, ce qui signifie que l'on ne passe jamais au second instant. Pour corriger ce processus, il faut ajouter un appel à **pause** à la fin de la boucle, afin qu'une seule itération de la boucle soit exécutée à chaque instant :

exemples/ch3/timer_ok.rml
```
let process timer delay s =
    ...
    then (emit s (); time := time');
    pause
  end
```

L'autre source de non-réactivité des programmes REACTIVEML est la récursivité :

exemples/ch3/instantaneous.rml
```
let rec process instantaneous s =
  emit s (); run (instantaneous s)
```

Ce processus boucle instantanément, tout comme dans l'exemple précédent. Là encore, on peut le corriger par un appel à **pause** avant l'appel récursif.

Nous définirons dans le chapitre 7 une *analyse de réactivité* qui détecte les processus (potentiellement) non-réactifs et affiche un avertissement dans le cas du processus `timer` :

```
Warning: This expression may be an instantaneous loop.
```

Cette analyse traite également le cas des processus récursifs, comme le processus `instantaneous` :

```
Warning: This expression may produce an instantaneous recursion.
```

Le principe de l'analyse est de vérifier qu'il se passe au moins un instant entre l'instanciation d'un processus et un appel récursif et que l'exécution du corps d'une boucle prend au moins un instant. On montrera que l'on peut également étendre cette analyse au cas des domaines

réactifs non coopératifs. Ainsi, l'analyse étendue affiche également un avertissement dans le cas du processus `nonreactive_domain` :

```
Warning: This reactive domain may not cooperate.
```

On a vu une première approche au problème des domaines non coopératifs : on considère qu'un processus qui n'attend jamais une horloge lente est incorrect et on crée une analyse pour détecter ce cas de figure. Il y a pourtant des processus utiles qui bouclent indéfiniment, comme par exemple le processus `sustain_v` (page 17) qui maintient un signal. Une approche alternative est donc de tenter de donner un sens à ces processus dans le contexte des domaines réactifs, ou au moins une façon sûre de les écrire. Nous allons pour cela ajouter une nouvelle instruction dans le langage : **`quiet pause`** `ck` attend le prochain instant de l'horloge `ck` mais doit être considéré comme en attente par le domaine réactif. L'intuition est qu'à la fin de chaque instant de son horloge locale `ck`, un domaine réactif organise une sorte de vote pour savoir s'il doit exécuter un autre instant local ou attendre son horloge parente. Un appel de **`pause`** `ck` revient à demander l'exécution d'un autre instant local. A l'opposé, un appel de **`quiet pause`** `ck` revient à ne pas voter. Un processus appelant **`quiet pause`** est donc passif et suit le rythme imposé par les autres processus contenus dans le domaine. L'appel de **`quiet pause`** `ck` termine dans le prochain instant de `ck`, mais le fait que cet instant s'exécute ou non dans l'instant courant de l'horloge parente du domaine dépend des autres processus du domaine.

Un exemple typique d'utilisation de cet opérateur est le maintien d'un signal sans rendre le domaine non coopératif :

exemples/ch3/quiet_sustain.rml

```
let process quiet_sustain_v s ck =
  loop emit s (last s); quiet pause ck end

let process hold_domain =
  domain ck do
    signal s default 0 gather (+) in
    run quiet_sustain_v s ck
    ||
    pause ck; emit s 3; pause global_ck; print_int (last s)
  done
```

La première branche du parallèle utilise l'opérateur **`quiet pause`**. Elle est donc passive et suit le rythme de la seconde branche. Puisque celle-ci appelle **`pause`** `ck` pendant le premier instant de `ck`, on doit exécuter un second instant de `ck`. Elle émet ensuite la valeur `3` sur `s`, puis attend le prochain instant de l'horloge globale. Comme plus aucun processus ne demande un autre instant local, le domaine réactif attend automatiquement le prochain instant de l'horloge globale avant d'exécuter son prochain instant local. Si on remplace **`quiet pause`** `ck` par **`pause`** `ck` dans la définition de `quiet_sustain_v`, alors le domaine devient non coopératif, ce que détecte l'analyse de réactivité.

Domaines et parallélisme

Bien que l'apport principal des domaines réactifs soit le gain de modularité du langage, ils permettent également de répondre à certaines problématiques liées à l'exécution parallèle et distribuée de REACTIVEML, qui est l'autre objectif de cette thèse :

Désynchronisation En introduisant des échelles de temps locales, les domaines réactifs permettent de distinguer les synchronisations locales de celles qui sont globales. En particulier, les instants de deux domaines réactifs de même horloge parente sont indépendants, comme on l'a vu dans le cas du processus `stutter` (page 33). On peut donc lancer ces deux domaines réactifs dans deux *threads* d'exécution en parallèle. Ils ne se synchroniseront alors qu'à la fin de chaque instant de leur horloge parente.

Absence de dépendance instantanée Il est interdit de dépendre instantanément d'un signal lent à l'intérieur d'un domaine réactif, comme dans le cas de `immediate_dep_wrong` (page 36). Cette limitation permet avant tout de garantir la cohérence du statut des signaux, c'est-à-dire que l'on ne va pas faire deux hypothèses contradictoires sur la présence d'un signal au cours d'un même instant. Elle a également un avantage pour l'exécution parallèle des programmes : l'exécution d'un domaine réactif au cours d'un instant de son horloge parente est indépendante des autres domaines réactifs et processus. On peut exécuter tous les instants de ce domaine en une fois. Les communications et synchronisations avec les autres processus et domaines se font toutes à la fin de l'instant de l'horloge parente.

Localité des signaux Un signal attaché à un domaine réactif ne peut pas être utilisé en dehors de ce domaine, puisque l'on ne peut pas voir les instants du domaine. Là encore, il s'agit d'une limitation liée à la sémantique des domaines réactifs, mais qui a aussi une conséquence pour l'exécution parallèle. En effet, si on exécute le domaine réactif dans un thread séparé, on sait qu'un signal attaché à ce domaine ne sera pas utilisé par les autres threads, puisqu'il ne peut pas s'échapper du domaine réactif. Il n'est donc pas nécessaire d'utiliser de mécanisme comme un verrou pour gérer les accès concurrents à ce signal. L'accès à ce signal est donc aussi efficace que dans un moteur d'exécution séquentiel.

Remplacer les références L'exemple de la banque (page 28) nous a montré que les programmes REACTIVEML actuels utilisent souvent des références pour effectuer plusieurs modifications par instant sur un état partagé. L'utilisation de signaux requiert en effet plusieurs instants, ce qui peut compliquer l'écriture des programmes comme on l'a montré à la fin du chapitre 2. C'est un problème puisque l'utilisation de références est incompatible avec une exécution parallèle ou distribuée du programme. Dans le cas de l'exécution parallèle, on perd les propriétés d'atomicité de modifications des références, ce qui rend leur utilisation très complexe. Dans le cas distribué, la notion de mémoire partagée disparait complètement. Les domaines réactifs permettent de s'affranchir totalement des références, puisque l'on peut masquer des instants liés à des communications locales.

3.2 Programmation avec les domaines réactifs

Maintenant que nous avons vu les principes des domaines réactifs, nous allons montrer comment les utiliser et quels problèmes ils permettent de résoudre. Nous reviendrons en particulier sur les différents exemples du chapitre 2.

Rendre un processus instantané On utilise ici un domaine réactif pour masquer les instants de calcul d'un processus qui ne communique pas avec le reste du programme. Ce processus devient alors instantané. Cela revient quasiment à lancer un programme REACTIVEML externe que l'on aurait préalablement compilé. Le processus `levelorder` (page 15) est un bon candidat pour cette transformation :

exemples/ch3/levelorder_inst.rml

```
let process levelorder_inst f t =
  domain ck do
    run levelorder f t
  done
```

Le domaine réactif d'horloge `ck` exécute plusieurs instants avant de terminer dans le même instant de l'horloge globale. Le processus `levelorder_inst` termine donc instantanément.

Masquer des communications locales On cherche cette fois-ci à masquer des communications locales. Un exemple typique est le processus `sig_incr_local` (page 26) que l'on a utilisé pour illustrer les problèmes de modularité du langage :

```
let process sig_incr s_in s_out =
  await s_in(v) in emit s_out (v + 2)
```

@ exemples/nbody/planets_heun.rml

```
let rec process body_heun env (x_t, v_t, w) =
  emit env (force (x_t, w));
  await env(f_t) in
  (* 1e etape *)
  let f_t = f_t x_t in
  let v_int = v_t ++. (dt **. f_t) in
  let x_int = x_t ++. (dt **. v_t) in
  (* 2e etape *)
  emit env (force (x_int, w));
  await env(f_int) in
  let f_int = f_int x_int in
  let v_tp = v_t ++. ((dt /. 2.0) **. (f_t ++. f_int)) in
  let x_tp = x_t ++. ((dt /. 2.0) **. (v_t ++. v_int)) in
  (* pas de temps suivant *)
  draw_body x_t x_tp w;
  pause global_ck;
  run body_heun env (x_tp, v_tp, w)

let process main =
  init_gui ();
  domain computation_ck do
    signal env default (fun _ -> zero_vector) gather add_force in
    for i = 1 to 100 dopar
      run body_heun env (random_planet ())
    done
    ||
    run body_heun env (sun ())
  done
```

FIGURE 3.1 – Simulation des n-corps avec la méthode d'Heun

```
let process sig_incr_domain s_in s_out =
  domain ck do
    signal tmp default 0 gather (+) in
    await s_in(v) in emit tmp (v+1)
    ||
    await tmp(v) in emit s_out (v+1)
  done
```

On utilise dans cet exemple un domaine réactif pour masquer les communications locales sur le signal `tmp`. Le calcul prend deux instants sur l'horloge locale `ck`, mais un seul sur l'horloge globale. Le processus `sig_incr_domain` est donc bien un raffinement de `sig_incr`.

Masquer des pas de calcul partagés par plusieurs processus On a vu deux cas où l'on utilisait des domaines réactifs de façon locale. On peut également utiliser un domaine réactif pour masquer des pas de calcul partagés par de multiples agents. On peut utiliser ce raisonnement pour étendre notre exemple de simulation du problème des n-corps du chapitre 2 (dont le code est visible sur la figure 2.1 page 20). On va maintenant utiliser une méthode numérique multi-pas pour résoudre le système d'équations différentielles et plus particulièrement la méthode d'Heun[3]. L'idée des méthodes multi-pas est de calculer plusieurs valeurs intermédiaires afin d'améliorer la précision

3. http ://en.wikipedia.org/wiki/Heun%27s_method

du calcul. Pour une équation de la forme $\dot{x} = f(x)$ à partir d'une date de départ t_0, une position initiale x_0 et un pas de simulation dt, on va calculer une suite de valeurs x_k^{int} et $\hat{x}_k \approx x(t_0 + k.dt)$:

$$x_{k+1}^{int} = \hat{x}_k + dt.f(\hat{x}_k)$$
$$\hat{x}_{k+1} = \hat{x}_k + \frac{dt}{2}\left(f(\hat{x}_k) + f(x_{k+1}^{int})\right)$$

La précision de cette méthode augmente quadratiquement avec la taille du pas de temps dt, alors que la progression n'est que linéaire dans le cas de la méthode d'Euler.

Le code REACTIVEML est visible sur la figure 3.1. Chaque pas de calcul correspond à un instant du domaine réactif d'horloge `computation_ck`. En effet, chaque étape du calcul nécessite le calcul de la force et donc une communication entre tous les processus sur le signal `env` d'horloge `computation_ck`. Chaque corps attend ensuite le prochain instant de l'horloge globale avant de procéder au calcul du pas de temps suivant. Le domaine attend donc automatiquement le prochain instant de l'horloge globale pour commencer le prochain pas de la simulation. On garde donc la correspondance entre un pas de temps de la simulation et un instant du programme. Le nombre de pas utilisés pour le calcul est invisible pour le reste du programme, par exemple responsable de l'affichage des planètes.

Subdiviser le pas de temps Les domaines réactifs permettent de masquer des instants liés à des calculs ou des synchronisations, comme on l'a vu dans les exemples précédents. Ils permettent également de subdiviser les instants afin de créer plusieurs échelles de temps. Par exemple, si un instant du programme correspond à une milliseconde dans la simulation, on peut créer un domaine réactif pour subdiviser chaque milliseconde en mille microsecondes.

La figure 3.2 montre un raffinement de l'exemple de capteur présenté dans la figure 2.2 du chapitre 2. On cherche à simuler plus précisément la consommation d'énergie du capteur, en prenant en compte que la consommation d'énergie dépend du nombre de messages envoyés. La radio du capteur est représentée par un petit processus (lignes 12 à 18) qui reçoit un message et un destinataire sur le signal `r_in`. On modélise l'envoi du message en attendant `packet_send_time` microsecondes pendant lesquelles la consommation d'énergie est augmentée de `send_power`. Ensuite, le message est envoyé au destinataire et on confirme l'envoi en émettant le signal `r_ack`.

On remarque que le domaine réactif d'horloge `us` contient un processus qui n'attend jamais d'horloge lente. Ce n'est pas un problème puisque le domaine est périodique. Dans ce cas, la valeur après **by** correspond au nombre d'instants qu'effectue le domaine et pas à une borne. Le domaine réactif d'horloge `us` effectue bien 1000 instants par instant de son domaine parent. On distingue ainsi deux cas d'utilisations des domaines réactifs :

- Soit les instants ont une signification comme dans cet exemple du capteur. La valeur après **by** représente alors le nombre d'instants que fait le domaine réactif à chaque instant de son horloge parente. On peut utiliser des processus qui bouclent sur l'horloge locale sans risque.
- Soit les instants servent seulement pour des communications et synchronisations internes, comme dans l'exemple des n-corps. Alors, la valeur après **by** représente une borne sur le nombre d'instants effectués par le domaine réactif. On peut omettre cette borne, mais il faut faire attention à ne pas créer un domaine non coopératif.

Ordonner des actions En permettant de créer une notion d'instant local, les domaines réactifs permettent d'ordonner des actions au cours d'un instant. On peut ainsi émuler les *micro-instants* de REACTIVEC [Bou91], mais de façon réellement modulaire. Nous reviendrons plus en détail sur cette comparaison dans la partie 5.3.

Supposons par exemple que l'on ait écrit un processus `noisy` qui affiche à chaque instant un résultat sur la sortie standard. Afin de mieux discerner les différents pas du programme, on souhaite ajouter deux lignes vides entre chaque instant. On souhaiterait logiquement écrire :

@ exemples/sensor/radio_energy.rml

```
let process node_with_energy me neighbors =
 domain us by 1000 do
  signal dead in
  signal energy default e_0 gather (fun x _ -> x) in
  signal power default 0.0 gather (+.) in
  signal r_in default (0,me) gather (fun x _ -> x) in
  signal r_ack in
  let process send msg n =
    emit r_in (msg, n); await immediate r_ack
  in
  ...
  do
    ... (* protocol *) ||
    loop (* radio *)
      await r_in (msg, n) in
      for i=1 to packet_send_time do
        emit power send_power; pause us
      done;
      emit n msg; emit r_ack
    end
    ||
    loop  (* power *)
      emit power on_power;
      if last energy < e_min
      then emit dead
      else emit energy (last energy -. (last power /. 1000.0));
      pause us
    end
  until dead done
 done
```

FIGURE 3.2 – Un capteur avec simulation de la consommation d'énergie

exemples/ch3/noisy_newline.rml

```
let process printer =
  run noisy
  ||
  loop print_newline (); print_newline (); pause end
```

Malheureusement cette solution ne fonctionne pas puisque rien ne garantit l'ordre d'exécution entre les deux branches du parallèle au cours de l'instant [4]. L'utilisation d'un domaine réactif permet de subdiviser l'instant et de garantir l'ordre d'exécution des deux processus :

exemples/ch3/noisy_newline.rml

```
let process printer_fixed =
  domain ck do
    run noisy
    ||
    loop
      pause ck;
      print_newline (); print_newline (); pause global_ck
    end
  done
```

4. Cette solution peut fonctionner pour un moteur d'exécution donné, mais rien ne garantit que ce soit le cas.

Pour garantir l'ordre entre les opérations, on exécute `noisy` dans le premier instant de l'horloge `ck`, puis la seconde branche du parallèle au cours du second instant. Pour que cette solution fonctionne, il faut que le processus `noisy` ne dépende pas de l'horloge locale. Par exemple, il ne faut pas qu'il appelle **`pause`** sans argument, puisque l'on change son horloge locale. Ce n'est pas une grande limitation puisqu'un appel à **`pause`** `ck` a le même comportement quelle que soit l'horloge locale : il attend le prochain instant de l'horloge `ck`.

Il existe en REACTIVEML un opérateur permettant d'obtenir le même résultat. Il s'agit de la composition parallèle séquentielle, notée `|>` et inspirée de l'opérateur `merge` des SUGARCUBES [BS98]. Ainsi, dans le cas de l'expression `e1 |> e2`, on exécute à chaque instant d'abord `e1`, puis `e2`. Nous ne l'avons pas inclus dans le langage puisque sa sémantique et son implémentation posent de nombreux problèmes, en particulier en présence de dépendances immédiates et dans le cas d'une exécution parallèle. D'un point de vue plus philosophique, nous pensons que la composition parallèle doit être commutative et associative, et que les dépendances doivent être explicites, soit avec des signaux soit avec une horloge locale.

3.3 Exemples

Agents avec domaines

Une nouvelle version de notre bibliothèque d'agents réactifs La mini-bibliothèque d'agents que nous avons décrite dans la partie 2.2 peut s'utiliser quasiment telle quelle avec les domaines réactifs. L'idée est d'utiliser un domaine réactif par agent, qui permet de masquer les pas de calcul locaux et rend possible le raffinement temporel. On peut également utiliser un domaine périodique afin de subdiviser les instants, comme on l'a vu dans l'exemple `node_with_energy`. On ajoute dans le type d'un agent l'horloge de communication entre les agents, qui est aussi l'horloge du signal de réception des messages :

@ exemples/sensor/agents_ck.rml

```
type ('msg, 'out) agent{'ck|} = {
    ...
    a_network_ck : ['ck];
  }
```

Le type `['ck]` désigne une horloge associée à la variable d'horloge `'ck`. On doit ajouter cette nouvelle variable de type dans la liste des paramètres du type `('msg, 'out) agent{'ck|}`. Nous expliquerons la signification de ces types et la syntaxe concrète plus en détail dans le chapitre 6 consacré au calcul d'horloges. On donne l'horloge `nck` de communication entre les agents en argument de la fonction `mk_agent` de création d'un agent :

@ exemples/sensor/agents_ck.rml

```
let process mk_agent name nck p =
  signal kill, msg_sig clock nck in
  let self = { a_name = name; a_msg_signal = msg_sig;
               a_kill_signal = kill; a_network_ck = nck } in
  let process agent args =
    do
      run p self args
    until kill done
  in
  agent, self
```

Il faut également modifier le processus d'envoi de messages. En effet, il faut que le signal envoyé avec le message et utilisé pour la réponse soit un signal lent, afin qu'il puisse échapper du domaine correspondant à l'agent. On écrit ainsi le code suivant (on peut voir l'ancienne définition de `send_and_wait` dans la figure 2.3 page 23) :

@ exemples/sensor/agents_ck.rml
```
let process send_and_wait o msg =
  signal s clock o.a_network_ck in
  emit o.a_msg_signal (Mreq (msg, s));
  await s([v]) in v
```

On peut remarquer que l'on ne peut plus utiliser la construction **await immediate one** pour récupérer la valeur du signal temporaire s. En effet, il s'agit d'une dépendance instantanée qui est interdite sur un signal lent comme s. On la remplace ici par un filtrage permettant de récupérer la seule valeur émise. La conséquence de ce changement est qu'il faut maintenant deux instants pour recevoir la réponse à une requête, contre un instant dans la version précédente de notre mini-bibliothèque d'agents.

La banque avec retrait L'utilisation des domaines réactifs pour la programmation des agents va permettre de corriger les problèmes de modularité que l'on a vus à la fin du chapitre précédent. On peut ainsi reprendre l'exemple de la banque avec retrait décrit page 27. On a vu que cet exemple utilisait un signal local pour séquentialiser les retraits, ce qui avait pour conséquence de demander deux instants pour simuler un pas de temps de la banque. On peut éviter cela en déclarant les signaux internes à l'intérieur d'un domaine réactif local :

@ exemples/bank/bank3_domain.rml
```
let process bank self () =
  domain lck do
    memory counter default 0 gather (fun f acc -> f acc) in
    signal req in
    ...
  done
```

On obtient ainsi un processus que l'on peut simuler en un instant de l'horloge globale.

Réinitialisation d'un signal Nous allons illustrer la réinitialisation des signaux à partir de l'exemple de la banque. Supposons que la banque souhaite désormais faire payer des frais à un client qui effectue plus de trois opérations par pas de temps. On va donc ajouter à l'état de la banque un signal à mémoire operations correspondant au nombre d'opérations effectuées, que l'on va incrémenter à chaque opération. On remet à zéro ce compteur d'opérations à chaque pas de temps en déclarant l'horloge globale comme horloge de réinitialisation du compteur :

exemples/bank/bank4.rml
```
let process bank self () =
  memory counter default 0 gather (fun f acc -> f acc) in
  memory operations default 0 gather (fun f acc -> f acc) reset global_ck in
  run react self
    begin proc msg return -> match msg with
    | Put x -> operations <<- (+) 1; counter <<- (+) x
    | Info -> return (last counter)
    end
  ||
  loop
    await operations(nb) in
    if nb > 3
    then (counter <<- (fun c -> c - 2); operations <<- (fun nb -> nb - 3))
  end
```

On rappelle que l'opérateur (<<-) est une version infixe de **emit**, ce qui implique que l'appel de operations <<- (+) 1 incrémente la valeur du signal à mémoire operations, qui est réinitialisé à zéro au début de chaque instant de l'horloge globale. Il compte donc bien le nombre d'opérations par pas de temps.

@ exemples/sensor/radio_refine.rml

```
(* Radio *)
let process radio self battery =
  run react self
    begin proc (msg, dst) return ->
      for i=1 to packet_send_time do
        battery << send_power; pause self.a_network_ck
      done;
      send dst msg;
      return ()
    end

(* Logiciel *)
let process software self (neighbors, radio) =
  run react self
    begin proc msg return ->
      if msg>1 then
        run iter_p (proc dst -> run send_and_wait radio (msg-1, dst)) neighbors
    end

(* Batterie *)
let process battery self sensor =
  memory energy clock self.a_network_ck
              default e_0 gather (fun f acc -> f acc) in
  run react self
    begin proc msg return ->
      emit energy (fun e -> e -. (msg /. 1000.0))
    end
  ||
  loop
    if last energy < e_min then kill sensor;
    pause self.a_network_ck
  end

(* Capteur complet *)
let process node self neighbors =
  domain us by 1000 do
    let battery_p, battery = run mk_agent "battery" us battery in
    let radio_p, radio = run mk_agent "radio" us radio in
    let soft_p, soft = run mk_agent "software" us software in
    run battery_p self || run radio_p battery || run soft_p (neighbors, radio)
    ||
    run react self
      begin proc msg return ->
        soft << msg
      end
  done
```

FIGURE 3.3 – Un capteur modélisé avec plusieurs agents

Exemple de raffinement L'utilisation de domaines réactifs dans la définition des agents permet également de programmer un agent comme la composition de plusieurs agents. Il s'agit bien de raffinement puisque le comportement extérieur de l'agent reste le même : les communications locales sont masquées par le domaine réactif de l'agent. Nous allons illustrer ce raffinement sur l'exemple du capteur avec modélisation de l'énergie de la figure 3.2. Chaque capteur est désormais composé de trois agents :

La radio (lignes 1-10) On modélise ici le composant matériel chargé de l'envoi des messages (on ne modélise pas la réception des messages). On reprend le code de la figure 3.2.

Le logiciel de contrôle (lignes 12-18) Cet agent modélise le logiciel embarqué sur le capteur. Il traite les messages entrants et envoie à la radio les messages sortants.

La batterie (lignes 20-32) La batterie reçoit des autres composants du système leur consommation énergétique et met à jour en conséquence sa réserve d'énergie. Le capteur ne fonctionne plus lorsque l'énergie passe en dessous du seuil `e_min`.

L'agent modélisant le capteur complet (lignes 34-46) est obtenu en lançant en parallèle ces trois composants à l'intérieur d'un domaine réactif. Il transmet les messages entrants à l'agent modélisant le logiciel de contrôle. Grâce aux domaines réactifs, cet agent est bien un raffinement de celui que l'on a montré page 21. On peut en particulier mélanger dans une même simulation des agents avec et sans modélisation de l'énergie. Le fait que l'agent soit modélisé par un seul agent ou la composition de plusieurs agents est complètement invisible.

Variations sur la simulation des n-corps

Changement de méthode d'intégration à la volée On a vu un peu plus tôt comment les domaines réactifs permettent d'utiliser des méthodes de résolution d'équations différentielles multi-pas comme la méthode d'Heun. Puisque l'on utilise un domaine réactif pour masquer les pas de calcul, on peut utiliser d'autres méthodes multi-pas et même changer de méthode à la volée. On commence par définir les différentes méthodes que l'on va utiliser et un signal global `current_imethod` indiquant la méthode à utiliser pour le calcul :

```
type imethod = Pause | Euler | EulerSemi | Heun | RK4
signal current_imethod default EulerSemi gather (fun x _ -> x)
```

On définit ensuite un processus de calcul pour chaque méthode, suivant le principe que l'on a vu pour la méthode d'Heun. On définit également une autre méthode qui ne fait rien, pour mettre en pause la simulation :

```
let process compute_pause env x_t v_t w =
  (x_t, v_t)
```

On définit enfin un processus `compute` qui appelle le processus correspondant à la méthode courante :

@ exemples/nbody/planets_switch.rml

```
let process compute env x_t v_t w =
  match last current_imethod with
  | Pause -> run compute_pause env x_t v_t w
  | Euler -> run compute_euler env x_t v_t w
  | EulerSemi -> run compute_euler_semi_implicit env x_t v_t w
  | Heun -> run compute_heun env x_t v_t w
  | RK4 -> run compute_rk4 env x_t v_t w

let rec process body env (x_t, v_t, w) =
  let x_tp, v_tp = run compute env x_t v_t w in
  draw_body x_t x_tp w;
  pause global_ck;
  run body env (x_tp, v_tp, w)
```

Le choix de la méthode peut se faire par exemple en écoutant les entrées clavier. On émet sur le signal `current_imethod` la méthode d'intégration à utiliser selon la touche pressée par l'utilisateur :

```
let process switch_method =
  loop
    if Graphics.key_pressed () then (
      let imethod = match Graphics.read_key () with
      | 'p' -> Pause
      | 'e' -> Euler
      | 's' -> EulerSemi
      | 'h' -> Heun
      | 'r' -> RK4
      | _ -> last current_imethod
      in
      emit current_imethod imethod
    );
    pause global_ck
  end
```

On remarque que ce processus est complètement indépendant du nombre de pas de la méthode de calcul. Il suffit d'appeler **pause** `global_ck`, puisque le calcul prend un instant de l'horloge globale quelle que soit la méthode de calcul.

Méthode d'intégration adaptative Une autre extension intéressante est d'utiliser une méthode adaptative pour résoudre le système d'équations différentielles. On utilise ici la méthode de Bogacki-Shampine [5] [BS89]. La figure 3.4 montre le code correspondant.

Le principe des méthodes adaptatives est de calculer en plusieurs pas à la fois les nouvelles positions mais aussi une estimation de l'erreur. Chaque corps émet son erreur sur le signal `error` (ligne 5) dont la fonction de combinaison calcule le maximum des valeurs émises. Ensuite, si l'erreur est supérieure à un seuil fixé (ligne 20), on recommence le calcul avec un pas plus petit. Pour cela, on émet sur le signal `do_step` le processus de calcul paramétré par le nouveau pas de temps (ligne 18). En réponse, chaque corps va interrompre son calcul actuel et le recommencer avec le nouveau pas de temps (grâce au combinateur `reset_every` (page 18)). Sinon, on émet le signal `commit`, ce qui signifie que le résultat obtenu pour ce pas de temps est le bon et que chaque corps peut donc attendre le pas de temps suivant.

On utilise pour cet exemple deux domaines réactifs imbriqués :

- L'horloge `computation_ck` correspond comme dans le cas de la méthode d'Heun aux différents pas de calcul.
- L'horloge `adapt_ck` correspond aux essais successifs de calcul. On peut a priori faire un nombre d'instants non borné de cette horloge. On utilise donc la propriété d'attente automatique des domaines réactifs pour gérer le passage à l'instant global suivant.

Comme pour les méthodes précédentes, chaque pas de temps correspond toujours à un seul instant de l'horloge globale.

5. http://en.wikipedia.org/wiki/Bogacki-Shampine_method

```
let rec process body env (do_step, error, commit)                @ exemples/nbody/planets_adapt.rml
          (x_t, v_t, w) =
  let rec process try_step compute =
    let x_tp, v_tp, e = run compute env x_t v_t w in
    emit error e;
    await commit;
    x_tp, v_tp
  in
  await do_step(compute) in
  let x_tp, v_tp = run reset_every do_step try_step compute in
  draw_body x_t x_tp w;
  pause global_ck;
  run body env (do_step, error, commit) (x_tp, v_tp, w)

let process error_checker (do_step, error, commit) =
  let rec process try_step s =
    emit do_step(compute_bogacki_shampine s);
    await error(e) in
    if e > error_threshold then (
      print_endline ("Error found; trying with step : "^
                     string_of_float (s /. 2.0));
      run try_step (s /. 2.0);
      run try_step (s /. 2.0)
    ) else
      emit commit
  in
  loop
    run try_step time_step;
    pause global_ck
  end

let process main =
  init_gui ();
  domain adapt_ck do
    signal error default 0.0 gather max in
    signal commit in
    domain computation_ck do
      signal do_step clock adapt_ck
          default compute_bogacki_shampine 0.0 gather (fun x _ -> x) in
      signal env default (fun _ -> zero_vector) gather add_force in
      run error_checker (do_step, error, commit)
      ||
      for i = 1 to 25 dopar
        run body env (do_step, error, commit) (random_planet ())
      done
      ||
      run body env (do_step, error, commit) (sun ())
    done
```

FIGURE 3.4 – Simulation des n-corps avec méthode adaptative

Deuxième partie

Sémantique

CHAPITRE 4

Sémantique comportementale

Nous allons maintenant présenter une sémantique formelle de REACTIVEML étendu avec les domaines réactifs. Nous nous appuierons sur un noyau du langage permettant d'encoder la majorité des constructions. Nous devrons ensuite poser un certain nombre de définitions et de notations que nous utiliserons pour définir les différentes sémantiques du langage. Nous définirons enfin formellement la sémantique *comportementale*, aussi dite *à grands pas*, du langage. Elle s'appuie sur les principes de la sémantique comportementale d'ESTEREL [BC84] dont elle reprend le nom et étend celle de REACTIVEML [MP08b]. Nous l'illustrerons sur plusieurs exemples puis nous discuterons certains choix.

4.1 Syntaxe abstraite du langage

Valeurs Les constantes c du langage sont définies comme en ML ou REACTIVEML par :

$$c ::= tt \mid ff \mid () \mid [] \mid \texttt{0} \mid \texttt{1} \mid \cdots \mid + \mid - \mid \ldots$$

Les noms d'horloges ck appartiennent à un ensemble dénombrable $\mathcal{C}$. On distingue deux valeurs particulières de $\mathcal{C}$. L'horloge $\top_{ck}$ est l'horloge globale de tous les programmes, alors que $\bot_{ck}$ est une horloge jamais activée, que l'on utilisera pour définir la réinitialisation des signaux. De la même façon, les noms de signaux n appartiennent à un ensemble dénombrable noté $\mathcal{N}$. Les valeurs v du langage sont les suivantes :

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

Une valeur est une constante, un nom de signal n^{ck} étiqueté par son horloge ck, un nom d'horloge ck, une paire de valeurs, une fonction ou un processus.

Expressions Dans la suite du manuscrit, nous allons utiliser la syntaxe abstraite suivante, qui est un noyau non minimal de notre extension de REACTIVEML :

$$\begin{aligned} e ::= {} & x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e \\ & \mid \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e, \texttt{d} : e, \texttt{g} : e, \texttt{rck} : e)\ \texttt{in}\ e \mid \texttt{emit}\ e\ e \mid \texttt{last}\ e \\ & \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{do}\ e\ \texttt{until}\ e(x) \to e \mid \texttt{do}\ e\ \texttt{when}\ e \\ & \mid \texttt{domain}\ x\ \texttt{by}\ e\ \texttt{do}\ e \mid \texttt{pause}\ e \mid \texttt{qpause}\ e \mid \texttt{local\_ck} \mid e\ \texttt{in}\ (ck, n/n) \end{aligned}$$

La base du langage est un λ-calcul avec appel par valeur, auquel on ajoute la définition de processus (`process`), le lancement de processus (`run`), la définition parallèle (`let`/`and`), la définition de signal (`signal`), l'émission sur un signal (`emit`) et la lecture de la dernière valeur d'un signal (`last`). Les structures de contrôle sont le test de présence (`present`/`then`/`else`), la préemption faible (`do`/`until`) et la suspension (`do`/`when`). Les expressions liées aux domaines réactifs sont

$$
\begin{aligned}
e_1 \;||\; e_2 &\triangleq \texttt{let \_} = e_1 \texttt{ and \_} = e_2 \texttt{ in } () \\
\texttt{let } x = e_1 \texttt{ in } e_2 &\triangleq \texttt{let } x = e_1 \texttt{ and \_} = () \texttt{ in } e_2 \\
\texttt{let } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 &\triangleq \texttt{let } f = \lambda x_1 \ldots . \lambda x_p . e_1 \texttt{ in } e_2 \\
\texttt{let rec } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 &\triangleq \texttt{let } f = (\texttt{rec } f = \lambda x_1 \ldots . \lambda x_p . e_1) \texttt{ in } e_2 \\
\texttt{let process } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 &\triangleq \texttt{let } f = \lambda x_1 \ldots . \lambda x_p . \texttt{process } e_1 \texttt{ in } e_2 \\
\texttt{let rec process } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 &\triangleq \texttt{let } f = (\texttt{rec } f = \lambda x_1 \ldots . \lambda x_p . \texttt{process } e_1) \texttt{ in } e_2 \\
e_1 ; e_2 &\triangleq \texttt{let \_} = e_1 \texttt{ in } e_2 \\
\texttt{loop } e &\triangleq \texttt{run } ((\texttt{rec } \mathit{loop} = \lambda x. \\
&\qquad\qquad \texttt{process } (\texttt{run } x ; \texttt{run } (\mathit{loop}\ x))) \ (\texttt{process } e)) \\
\texttt{emit } e &\triangleq \texttt{emit } e\ () \\
\texttt{await immediate } e &\triangleq \texttt{do } () \texttt{ when } e
\end{aligned}
$$

FIGURE 4.1 – Expressions dérivées du noyau

la création d'un domaine réactif (`domain`), l'attente du prochain instant d'une horloge (`pause`), l'attente silencieuse (`qpause`, qui est notée **`quiet pause`** `e` dans la syntaxe concrète) et la lecture de l'horloge locale (`local_ck`). La dernière expression e_1 `in` $(ck', i/j)$ représente un domaine réactif en cours d'exécution. Elle ne s'obtient que par instanciation d'un domaine `domain` x `by` e_b `do` e_1 et ne peut pas apparaître dans un programme. ck' représente l'horloge du nouveau domaine réactif dans lequel on doit exécuter le corps e_1. i représente le nombre d'instants effectués dans l'instant courant de l'horloge parente et j le nombre maximal d'instants par instant de l'horloge parente.

La déclaration d'un signal (`signal`) est suivie d'un booléen indiquant s'il s'agit d'un signal à mémoire (`h`), de l'horloge du signal (`ck`), de sa valeur par défaut (`d`), de sa fonction de combinaison (`g`) et enfin de son horloge de réinitialisation (`rck`).

Expressions dérivées Nous pouvons encoder de nombreuses constructions du langage à partir de ce noyau. La figure 4.1 montre quelques exemples de ces encodages, qui sont identiques à ceux que l'on peut voir dans [MP08b]. On note _ lorsque la variable n'apparait pas libre dans le corps du `let`. Par exemple, on peut encoder la séquence ou l'opérateur de composition parallèle synchrone à partir de la définition parallèle `let`/`and`. De même, on obtient la boucle inconditionnelle à l'aide d'un processus récursif. On peut remarquer au passage qu'il n'y a pas vraiment de notion de processus récursif et qu'il s'agit plutôt d'une fonction récursive dont le résultat est un processus.

Plusieurs expressions dérivées sont différentes dans le langage étendu. La première est bien entendu la déclaration de signal, puisque l'on a ajouté plusieurs paramètres :

$$\texttt{signal } s \texttt{ in } e \triangleq \texttt{signal } s\ (\texttt{h} : \mathit{ff}, \texttt{ck} : \texttt{local\_ck}, \texttt{d} : [], \texttt{g} : (\lambda x. \lambda l. x :: l), \texttt{rck} : \bot_{ck}) \texttt{ in } e$$

L'horloge par défaut d'un signal est l'horloge locale au moment de sa définition. Sa valeur est la liste des valeurs émises sur le signal. Sa valeur par défaut est donc la liste vide [] et sa fonction de combinaison est la concaténation notée ::. Enfin, il n'est par défaut jamais réinitialisé, ce que l'on indique en utilisant l'horloge $\bot_{ck}$. La définition d'un signal à mémoire, de l'horloge du signal ou de son horloge de réinitialisation se fait de la façon suivante :

$$
\begin{aligned}
&\texttt{memory } m \texttt{ clock } e_{ck} \texttt{ default } e_d \texttt{ gather } e_g \texttt{ reset } e_{rck} \texttt{ in } e \\
&\qquad\qquad \triangleq \texttt{signal } m\ (\texttt{h} : tt, \texttt{ck} : e_{ck}, \texttt{d} : e_d, \texttt{g} : e_g, \texttt{rck} : e_{rck}) \texttt{ in } e
\end{aligned}
$$

La seconde nouveauté concerne l'attente d'un signal (`await`) que l'on encode comme en REACTIVEML par la préemption d'une boucle infinie. Dans le langage étendu, il faut que cette

boucle utilise l'attente silencieuse avec l'opérateur `qpause` afin de ne pas rendre le domaine correspondant non coopératif, c'est-à-dire équivalent à une boucle infinie instantanée :

$$\texttt{await}\ e_s(x)\ \texttt{in}\ e_1 \triangleq \texttt{do}\ (\texttt{loop}\ (\texttt{qpause}\ \texttt{local\_ck}))\ \texttt{until}\ e_s(x) \to e_1$$

Enfin, l'appel à `pause` sans argument attend le prochain instant de l'horloge locale :

$$\texttt{pause} \triangleq \texttt{pause}\ \texttt{local\_ck}$$
$$\texttt{qpause} \triangleq \texttt{qpause}\ \texttt{local\_ck}$$

Contexte d'évaluation On définit un *contexte d'évaluation* Γ comme une expression contenant un « trou » noté $[]$. Il est défini par la grammaire suivante :

$$\begin{aligned}
\Gamma ::= {} & [] \mid \Gamma\ e \mid e\ \Gamma \mid (\Gamma, e) \mid (e, \Gamma) \mid \texttt{run}\ \Gamma \\
& \mid \texttt{let}\ x = \Gamma\ \texttt{and}\ x = e\ \texttt{in}\ e \mid \texttt{let}\ x = e\ \texttt{and}\ x = \Gamma\ \texttt{in}\ e \\
& \mid \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : \Gamma, \texttt{d} : e, \texttt{g} : e, \texttt{rck} : e)\ \texttt{in}\ e \\
& \mid \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e, \texttt{d} : \Gamma, \texttt{g} : e, \texttt{rck} : e)\ \texttt{in}\ e \\
& \mid \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e, \texttt{d} : e, \texttt{g} : \Gamma, \texttt{rck} : e)\ \texttt{in}\ e \\
& \mid \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e, \texttt{d} : e, \texttt{g} : e, \texttt{rck} : \Gamma)\ \texttt{in}\ e \\
& \mid \texttt{emit}\ \Gamma\ e \mid \texttt{emit}\ e\ \Gamma \mid \texttt{last}\ \Gamma \\
& \mid \texttt{present}\ \Gamma\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{do}\ e\ \texttt{until}\ \Gamma(x) \to e \mid \texttt{do}\ \Gamma\ \texttt{until}\ v(x) \to e \mid \texttt{do}\ e\ \texttt{when}\ \Gamma \\
& \mid \texttt{domain}\ x\ \texttt{by}\ \Gamma\ \texttt{do}\ e \mid \texttt{pause}\ \Gamma \mid \texttt{qpause}\ \Gamma
\end{aligned}$$

4.2 Notations

Les notations que nous allons définir dans cette partie seront utilisées à la fois pour la définition formelle de la sémantique comportementale dans la partie suivante, mais aussi pour la sémantique opérationnelle du chapitre 5. Un index rappelant les pages où sont définies ces notations est disponible à la fin du manuscrit (page 233).

Substitution

On note $e[x \leftarrow e_2]$ la substitution modulo α-équivalence, c'est-à-dire la substitution classique du λ-calcul, qui remplace toutes les occurrences libres de la variable x par l'expression e_2 dans e. Par exemple, on a :

$$((\lambda x.e_1)\ x)[x \leftarrow 0] = (\lambda x.e_1)\ 0$$

Fonctions partielles

Dans la suite, on note $\{\}$ une fonction partielle de domaine vide et $\{v_1 \mapsto v_2\}$ la fonction partielle définie uniquement en v_1 et qui associe v_2 à v_1. Étant données deux fonctions partielles $f, g : A \rightharpoonup B$, on étend une opération d'union $\sqcup : B \times B \to B$ aux fonctions partielles de A dans B. On note $f \sqcup g : A \rightharpoonup B$, qui est définie par :

$$\forall x \in Dom(f) \cup Dom(g).\ (f \sqcup g)(x) = \begin{cases} f(x) \sqcup g(x) & \text{si } x \in Dom(f) \text{ et } x \in Dom(g) \\ f(x) & \text{si } x \in Dom(f) \text{ et } x \notin Dom(g) \\ g(x) & \text{si } x \notin Dom(f) \text{ et } x \in Dom(g) \end{cases}$$

Le domaine de la fonction partielle $f \sqcup g$ est l'union des domaines de f et g. L'intuition de cette opération est de prendre en chaque point x l'union de $f(x)$ et de $g(x)$. De la même façon, on définit

l'intersection de deux fonctions partielles, notée $f \sqcap g$, comme l'intersection point-à-point. Enfin, on note $g = f[v_1 \mapsto v_2]$ la fonction définie par :

$$\forall x \in Dom(f) \cup \{v_1\}.\ g(x) = \begin{cases} v_2 & \text{si } x = v_1 \\ f(x) & \text{si } x \in Dom(f) \text{ et } x \neq v_1 \end{cases}$$

Pile d'exécution

Une *pile d'exécution* s est une liste de couples (ck, i) qui associe à une horloge ck l'indice i de l'instant actuel de cette horloge. Une pile ne peut pas contenir deux fois la même horloge. On note $[]$ la pile vide. Si s est une pile, alors $(ck, i) :: s$ et $s :: (ck, i)$ sont aussi des piles. Dans le premier cas, on ajoute ck en bas de la pile, alors que dans le second cas on l'ajoute au sommet de la pile. On note $\texttt{top}(s)$ l'horloge au sommet de la pile. Ainsi, on a par exemple $\texttt{top}(s :: (ck, i)) = ck$. On note $Clocks(s)$ l'ensemble des horloges apparaissant dans s.

Une pile désigne un instant de l'horloge au sommet de la pile, qui est par convention l'horloge locale. Ainsi, la pile $(\top_{ck}, 2) :: (ck, 0)$ désigne le premier instant de l'horloge ck inclus dans le troisième instant de l'horloge globale $\top_{ck}$.

Ordre induit par une pile Une pile s induit un ordre sur les horloges qu'elle contient que l'on note $\preceq_s$. Le sommet de la pile $\texttt{top}(s)$ est le minimum de $Clocks(s)$ pour cet ordre. On note également $\min_s(ck_1, ck_2)$ le minimum associé. Ainsi, le minimum de l'horloge locale $\texttt{top}(s)$ et d'une autre horloge ck apparaissant dans la pile est toujours égal à $\texttt{top}(s)$:

$$\forall ck \in Clocks(s).\ \texttt{top}(s) \preceq_s ck \ \wedge\ \min_s(ck, \texttt{top}(s)) = \texttt{top}(s)$$

L'ordre induit par la pile correspond aux relations de descendance dans l'arbre d'horloges. Ainsi, $ck_1 \preceq_s ck_2$ signifie que ck_1 est plus rapide que ck_2.

Restriction d'une pile En plus de désigner un instant de l'horloge au sommet de la pile, une pile permet d'accéder à l'instant courant de ses horloges parentes grâce à une opération que l'on appelle la restriction d'une pile s à une horloge ck, qui est notée $s_{\restriction ck}$ et définie par :

$$(s :: (ck', i))|_{ck} \triangleq \begin{cases} s :: (ck', i) & \text{si } ck' = ck \\ s_{\restriction ck} & \text{sinon} \end{cases}$$

Le principe de cette opération est d'enlever le haut de pile pour que (ck, i) devienne le sommet de la pile. On obtient ainsi une nouvelle pile désignant l'instant courant de l'horloge ck. Cette opération n'est pas définie si ck n'apparait pas dans la pile.

Équivalence et ordre entre piles Deux piles sont équivalentes, ce que l'on note $s_1 \equiv s_2$, si elles désignent des instants d'une même horloge, c'est-à-dire si elles contiennent les mêmes noms d'horloges dans le même ordre :

$$[] \equiv [] \qquad\qquad s_1 \equiv s_2 \Rightarrow (ck, i_1) :: s_1 \equiv (ck, i_2) :: s_2$$

On définit également une relation d'ordre $\leq$ sur les piles, à ne pas confondre avec l'ordre $\preceq_s$ induit par une pile s donnée. On a $s_1 \leq s_2$ si les deux piles sont équivalentes et que s_1 correspond à un instant antérieur à celui associé à s_2. On utilise pour cela un ordre lexicographique sur les indices de la pile :

$$[] \leq [] \qquad (ck, i_1) :: s_1 \leq (ck, i_2) :: s_2 \quad \Leftrightarrow \quad (i_1 < i_2 \wedge s_1 \equiv s_2) \vee (i_1 = i_2 \wedge s_1 \leq s_2)$$

On note $<$ l'ordre strict associé. On peut remarquer que l'on ne compare que des piles équivalentes, c'est-à-dire contenant les mêmes noms d'horloges, et que deux piles équivalentes sont toujours comparables.

Ensemble de piles bien formé Intuitivement, un ensemble de piles est bien formé s'il contient les instants successifs de toutes les horloges apparaissant dans un arbre d'horloges. Formellement, l'ensemble de piles S est bien formé si :

– Toutes les piles de même sommet sont équivalentes :

$$\forall s_1, s_2 \in S.\ \texttt{top}(s_1) = \texttt{top}(s_2) \Rightarrow s_1 \equiv s_2$$

– Il n'y a pas de « trou » dans les indices des piles de même sommet :

$$s :: (ck, i) \in S \ \wedge\ 0 \leq j \leq i \quad \Rightarrow \quad s :: (ck, j) \in S$$

Les ensembles de piles que nous manipulerons par la suite seront toujours bien formés. Puisque toutes les piles de même sommet d'un tel ensemble sont comparables, on peut définir le minimum des piles de sommet ck, que l'on appelle le premier instant de ck dans S et que l'on note $\texttt{first}(ck, S)$. On définit de la même manière le maximum de ces piles, que l'on appelle dernier instant et note $\texttt{last}(ck, S)$:

$$\begin{aligned}\texttt{first}(ck, S) &\triangleq \min\{s \in \mathcal{S} \mid \texttt{top}(s) = ck\}\\ \texttt{last}(ck, S) &\triangleq \max\{s \in \mathcal{S} \mid \texttt{top}(s) = ck\}\end{aligned}$$

On définit également le successeur de s dans S, noté $\texttt{succ}(s, S)$, comme le plus petit majorant de s dans S. Il s'agit de la pile désignant l'instant suivant de l'horloge $\texttt{top}(s)$. Cette opération n'est pas définie pour le maximum des piles d'une horloge, c'est-à-dire si $s = \texttt{last}(\texttt{top}(s), S)$.

Par exemple, l'ensemble de piles S suivant est bien formé :

$$S \triangleq \{(\top_{ck}, 0),\ (\top_{ck}, 1) :: (ck, 0),\ (\top_{ck}, 1) :: (ck, 1),\ (\top_{ck}, 2) :: (ck, 0)\}$$

Le premier instant de ck dans S est $\texttt{first}(ck, S) = (\top_{ck}, 1) :: (ck, 0)$, alors que le dernier est $\texttt{last}(ck, S) = (\top_{ck}, 2) :: (ck, 0)$. On a également $\texttt{succ}(\texttt{first}(ck, S), S) = (\top_{ck}, 1) :: (ck, 1)$ et $\texttt{succ}((\top_{ck}, 1) :: (ck, 1), S) = (\top_{ck}, 2) :: (ck, 0)$.

Instant suivant d'une pile On définit l'instant suivant d'une pile étant donné un ensemble C d'horloges en fin d'instant, noté $\texttt{snext}(s, C)$, par :

$$\texttt{snext}(s :: (ck, i), \{ck\} \cup C) = \begin{cases} s :: (ck, i+1) & \text{si } C = \emptyset \\ \texttt{snext}(s, C) :: (ck, 0) & \text{sinon} \end{cases}$$

L'idée de cette opération est d'incrémenter le compteur de l'horloge la plus lente de C, puis de remettre à zéro les compteurs des horloges plus rapides. La pile $\texttt{snext}(s, C)$ désigne donc le premier instant de l'horloge $\texttt{top}(s)$ inclus dans le prochain instant de toutes les horloges contenues dans C.

Ensemble de noms

Afin de pouvoir s'assurer de la fraîcheur des noms apparaissant dans les règles, par exemple pour la définition d'un nouveau signal, on utilise un ensemble de noms noté N. Il s'agit d'une paire constituée d'un ensemble N^n de noms de signaux et d'un ensemble N^{ck} de noms d'horloges. On note $N_1 \cdot N_2$ l'union disjointe de deux ensembles de noms, qui est faite pour les deux composantes :

$$N_1 \cdot N_2 = N_1^n \cup N_2^n, N_1^{ck} \cup N_2^{ck} \quad \text{si} \quad N_1^n \cap N_2^n = \emptyset \text{ et } N_1^{ck} \cap N_2^{ck} = \emptyset$$

On note également $N \cdot \{n\}$ pour $N \cdot (\{n\}, \emptyset)$ et de même $N \cdot \{ck\}$ pour $N \cdot (\emptyset, \{ck\})$. Comme il s'agit d'une union disjointe, cette opération n'est définie que si $n \notin N^n$ et $ck \notin N^{ck}$.

Environnement de signaux

L'environnement de signaux contient toutes les informations relatives aux signaux, comme par exemple leurs valeurs ou leurs fonctions de combinaison. Nous allons maintenant définir la notion d'environnement local de signaux, qui correspond à la définition d'un environnement de signaux en REACTIVEML [MP08b]. Nous l'étendons pour prendre en compte les nouveaux traits associés aux signaux, c'est-à-dire les signaux à mémoire et la réinitialisation.

Définition 1. On appelle *environnement local de signaux* S une fonction partielle qui associe à un nom de signal un n-uplet (d, g, l, m, h, rck) où d est la valeur par défaut du signal, g sa fonction de combinaison, l sa valeur précédente, m est le multi-ensemble des valeurs émises au cours de l'instant courant, h indique s'il s'agit d'un signal à mémoire et rck est son horloge de réinitialisation.

On note $S^d(n)$ le champ d associé au signal n dans l'environnement local de signaux S, c'est-à-dire sa valeur par défaut, et de même pour les autres champs ($S^g(n), S^l(n), S^m(n), S^h(n), S^{rck}(n)$). Si le signal n a le type (τ_1, τ_2) `event`, alors ces différents champs ont les types suivants (on note τ_i les types du langage qui seront définis plus précisément dans le chapitre 6) :

$$S^d(n) : \tau_2 \qquad S^g(n) : \tau_1 \to \tau_2 \to \tau_2 \qquad S^l(n) : \tau_2$$
$$S^m(n) : \tau_1 \texttt{ multiset} \qquad S^h(n) : \texttt{bool} \qquad S^{rck}(n) : \texttt{clock}$$

On note également $n \in S$ si le signal n est présent, c'est-à-dire si le multi-ensemble des valeurs émises est non vide, ou autrement dit si $S^m(n) \neq \emptyset$. On note $n \notin S$ dans le cas contraire. On définit aussi la valeur d'un signal, notée $S^v(n)$, qui est obtenue en itérant la fonction de combinaison sur les valeurs émises, en partant de la valeur précédente ou de la valeur par défaut selon qu'il s'agit ou non d'un signal à mémoire :

$$S^v(n) \triangleq \begin{cases} \mathit{fold}\ S^g(n)\ S^m(n)\ S^d(n) & \text{si } S^h(n) = \mathit{ff} \\ \mathit{fold}\ S^g(n)\ S^m(n)\ S^l(n) & \text{si } S^h(n) = \mathit{tt} \end{cases}$$

$$\text{où} \quad \mathit{fold}\ f\ (\{v_1\} \uplus m)\ v_2 = \mathit{fold}\ f\ m\ (f\ v_1\ v_2)$$
$$\mathit{fold}\ f\ \emptyset\ v_2 = v$$

Pour la sémantique du langage étendu, on doit conserver les environnements locaux de signaux correspondant à tous les instants des horloges rapides inclus dans un instant de l'horloge globale. On associe donc un environnement local de signaux à chaque pile, qui définit bien de façon unique un instant de l'horloge au sommet de la pile :

Définition 2. Un *environnement de signaux* $\mathcal{S}$ est une fonction partielle des piles dans les environnements locaux de signaux.

On note $\mathit{Sig}(\mathcal{S}) = \bigcup_{s \in Dom(\mathcal{S})} Dom(\mathcal{S}(s))$ l'ensemble des noms de signaux apparaissant dans un environnement de signaux et $\mathit{Clocks}(\mathcal{S}) = \bigcup_{s \in Dom(\mathcal{S})} \mathit{Clocks}(s)$ l'ensemble des horloges apparaissant dans les piles du domaine de $\mathcal{S}$. On étend également toutes les notations des environnements locaux de signaux. Par exemple, on note $\mathcal{S}^d(s)(n) \triangleq (\mathcal{S}(s))^d(n)$.

Environnement de signaux bien formé Un environnement de signaux est bien formé si les environnements locaux de signaux correspondant à des instants successifs d'une même horloge sont bien compatibles entre eux. Cela signifie par exemple que les valeurs par défaut des signaux sont conservées, ou encore que la dernière valeur $S^l(n)$ de chaque signal est bien la dernière valeur émise sur ce signal.

Définition 3. Un environnement de signaux $\mathcal{S}$ est bien formé si l'ensemble de piles $Dom(\mathcal{S})$ est bien formé et si, pour toute horloge $ck \in \mathit{Clocks}(\mathcal{S})$ et toute pile s vérifiant $\texttt{first}(ck, Dom(S)) \leq s$

et $s < \texttt{last}(ck, Dom(S))$, on a les propriétés suivantes, avec $s' = \texttt{succ}(s, Dom(S))$:

$$\mathcal{S}^d(s') \subseteq \mathcal{S}^d(s) \text{ et } \mathcal{S}^g(s') \subseteq \mathcal{S}^g(s) \text{ et } \mathcal{S}^{rck}(s') \subseteq \mathcal{S}^{rck}(s)$$

$$\forall n \in Dom(\mathcal{S}(s)).\ \mathcal{S}^l(s')(n) = \begin{cases} \mathcal{S}^d(s)(n) & \text{si } s_{|\mathcal{S}^{rck}(s)(n)} \neq s'|_{\mathcal{S}^{rck}(s)(n)} \\ \mathcal{S}^v(s)(n) & \text{sinon et que } n \in \mathcal{S}(s) \\ \mathcal{S}^l(s)(n) & \text{sinon et que } n \notin \mathcal{S}(s) \end{cases}$$

On voit que la valeur par défaut $\mathcal{S}^d$, la fonction de combinaison $\mathcal{S}^g$ et l'horloge de réinitialisation $\mathcal{S}^{rck}$ doivent être conservées d'un instant à l'autre. Pour la dernière valeur $\mathcal{S}^l(s')(n)$, elle est soit égale à la valeur par défaut si jamais la nouvelle pile s' correspond à un nouvel instant de l'horloge $\mathcal{S}^{rck}(s)(n)$ de réinitialisation du signal (première ligne), soit à la valeur émise au cours de l'instant précédent (deuxième ligne), soit à la dernière valeur émise à un instant précédent (troisième ligne).

Successeur d'un environnement local de signaux On définit le successeur d'un environnement local de signaux S étant donné un ensemble C d'horloges en fin d'instant, noté $\texttt{next}_C(S)$, comme le plus petit environnement local de signaux vérifiant :

$$\forall n \in Dom(S).\ \texttt{next}_C(S)(n) = \begin{cases} (S^d(n), S^g(n), S^d(n), \emptyset, S^h(n), S^{rck}(n)) & \text{si } S^{rck}(n) \in C \\ (S^d(n), S^g(n), S^v(n), \emptyset, S^h(n), S^{rck}(n)) & \text{si } S^{rck}(n) \notin C \text{ et } n \in S \\ (S^d(n), S^g(n), S^l(n), \emptyset, S^h(n), S^{rck}(n)) & \text{si } S^{rck}(n) \notin C \text{ et } n \notin S \end{cases}$$

Le but de cette opération est de calculer l'environnement de signaux à utiliser à l'instant suivant. Il faut donc calculer la nouvelle valeur précédente de tous les signaux et remettre à zéro l'ensemble des valeurs émises. Les autres champs sont conservés tels quels. La dernière valeur du signal est égale à la valeur par défaut si on est dans la fin de l'instant de l'horloge S^{rck} de réinitialisation du signal (première ligne), soit à la valeur émise à l'instant précédent (deuxième ligne) ou bien à l'ancienne valeur précédente si le signal est absent (troisième ligne). Si on omet l'horloge de réinitialisation d'un signal, il ne sera jamais réinitialisé puisque l'horloge $\bot_{ck}$ n'apparaitra jamais dans l'ensemble C.

Prédicat de fin d'instant Le prédicat $\texttt{eoi}_{s,\mathcal{S}}(ck)$ signifie que la pile s décrit le dernier instant de l'horloge locale $\texttt{top}(s)$ inclus dans l'instant courant de l'horloge ck dans l'environnement de signaux $\mathcal{S}$. Autrement dit, le prédicat est vrai si le prochain instant de $\texttt{top}(s)$ est inclus dans le prochain instant de ck. En particulier, il est toujours vérifié pour l'horloge locale $\texttt{top}(s)$. Il est défini formellement par :

$$\frac{s = \texttt{last}(\texttt{top}(s), Dom(\mathcal{S}))}{\texttt{eoi}_{s,\mathcal{S}}(ck)} \qquad \frac{s_{|ck} \neq (\texttt{succ}(s, Dom(\mathcal{S})))|_{ck}}{\texttt{eoi}_{s,\mathcal{S}}(ck)}$$

Le prédicat n'est pas défini si l'horloge ck n'apparait pas dans la pile.

Événements

Un *événement* associe à un signal les valeurs émises sur ce signal. Nous reprenons la même structure que pour les environnements de signaux.

Définition 4. Un *événement local* E est une fonction partielle des noms de signaux dans les multi-ensembles de valeurs.

Définition 5. Un *événement* $\mathcal{E}$ est une fonction partielle des piles dans les événements locaux.

On définit l'ajout d'un événement $\mathcal{E}$ à un environnement de signaux $\mathcal{S}$, que l'on note $\mathcal{S} + \mathcal{E}$, comme l'environnement de signaux dans lequel on a ajouté l'ensemble des valeurs contenues

dans $\mathcal{E}$ aux multi-ensembles des valeurs émises dans $\mathcal{S}$. Autrement dit, pour tout s dans $Dom(\mathcal{E})$ et tout n dans $Dom(\mathcal{E}(s))$:

$$(\mathcal{S}+\mathcal{E})^m(s)(n) \triangleq \begin{cases} \mathcal{S}^m(s)(n) \sqcup \mathcal{E}(s)(n) & \text{si } s \in Dom(\mathcal{S}) \text{ et } n \in Dom(\mathcal{S}(s)) \\ \mathcal{E}(s)(n) & \text{sinon} \end{cases}$$

Statut

Dans la sémantique comportementale de REACTIVEML [MP08b], chaque réaction est caractérisée par un booléen (noté b) qui indique si l'expression obtenue a terminé son exécution, c'est-à-dire est une valeur, ou bien si elle attend l'instant suivant. Dans la sémantique étendue, ce *statut* k indique maintenant si l'expression a terminé son exécution, ce que l'on note 0, ou l'horloge dont elle attend le prochain instant :

$$k \triangleq 0 \mid ck$$

On définit le minimum sur les statuts comme l'extension du minimum sur les horloges en ajoutant 0 comme maximum :

$$\min_s(0,0) = 0 \qquad \min_s(0,ck) = \min_s(ck,0) = ck \qquad \min_s(ck,ck') = ck \quad \text{si } ck \preceq_s ck'$$

On rappelle que le minimum de deux horloges est l'horloge la plus rapide, qui est ici ck. On définit de façon similaire le maximum $\max_s(k_1,k_2)$.

4.3 Sémantique comportementale

Nous allons maintenant présenter formellement les règles définissant la sémantique comportementale du langage.

Relation

L'exécution d'un programme est définie comme une succession potentiellement infinie de réactions, correspondant aux instants successifs de l'horloge globale $\top_{ck}$. Elle se termine lorsque le statut k est égal à 0. Pendant le ième instant, le programme e_i lit des entrées I_i, émet des sorties O_i et se réécrit en e'_i. L'expression e'_i est le résidu à exécuter à l'instant suivant, ce qui signifie que $e_{i+1} = e'_i$.

$$e_i \xrightarrow[O_i,k]{I_i} e'_i$$

Pour définir cette relation, on utilise la réaction d'une expression qui est donnée par :

$$N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'$$

qui signifie que dans la pile s et l'environnement de signaux $\mathcal{S}$, l'expression e se réécrit en e' avec le statut k, émet les signaux dans $\mathcal{E}$ et utilise les noms de signaux et d'horloges contenus dans N. La sémantique d'un instant du programme est donnée par :

$$e_i \xrightarrow[O_i,k]{I_i} e'_i \quad \Leftrightarrow \quad \mathcal{S}_i \text{ est le plus petit environnement de signaux (au sens de l'inclusion) tel que} \quad \begin{cases} N_i, (\top_{ck}, i) \vdash e_i \xrightarrow[\mathcal{S}_i]{\mathcal{E}_i,k_i} e'_i & \\ O_i \subseteq \mathcal{S}_i^v((\top_{ck}, i)) & (1) \\ \mathcal{S}_i^m = \mathcal{E}_i \sqcup \{(\top_{ck}, i) \mapsto I_i\} & (2) \\ \bigsqcup_{j \le i} \mathcal{S}_j \text{ bien formé} & (3) \end{cases}$$

Les conditions supplémentaires ont la signification suivante :

(1) Les sorties du programme sont incluses dans l'environnement de signaux. Plus précisément, elles sont contenues dans l'environnement local de signaux correspondant à l'horloge globale $\top_{ck}$, c'est-à-dire qu'elles ne varient qu'à chaque instant de l'horloge globale.

(2) L'environnement de signaux $\mathcal{S}_i$ contient exactement les valeurs émises pendant l'instant et les entrées du programme. Autrement dit, l'ensemble $\mathcal{S}^m(n^{ck})$ des valeurs émises pour un signal correspond aux valeurs émises contenues dans $\mathcal{E}(n^{ck})$ auxquelles il faut ajouter les entrées I_i du programme. La valeur du signal est donc obtenue en combinant toutes les valeurs émises sur ce signal pendant un instant de son horloge.

(3) La dernière condition exprime le fait que les environnements locaux de signaux associés à la même horloge sont bien cohérents entre eux. Cela signifie que les valeurs par défaut, fonctions de combinaison et horloges de réinitialisation de tous les signaux sont conservées d'un instant sur l'autre et que les valeurs précédentes sont bien définies.

Remarque 2. Afin de définir l'horloge globale, la sémantique d'un programme p est en fait donnée par la réduction de l'expression $\tilde{p}$ définie par :

$$\tilde{p} \triangleq \texttt{let}\ global\_ck = \texttt{local\_ck}\ \texttt{in}\ p$$

On voit que $global\_ck$ est une variable du programme, alors que `local_ck` est un mot-clé du langage dont la valeur dépend du contexte.

Règles

Les règles définissant la sémantique comportementale du langage étendu sont données dans les figures 4.2, 4.3 et 4.4. Par convention, ck désigne l'horloge locale, c'est-à-dire $\texttt{top}(s)$. ck' désigne l'horloge des signaux, qui doit être plus lente ou égale à l'horloge locale, ainsi que l'horloge locale des domaines réactifs.

Expressions de base La figure 4.2 montre les règles de la sémantique des expressions de base du langage. La plupart des règles sont similaires à celles de REACTIVEML données dans [MP08b]. On peut tout de même faire plusieurs commentaires :

- L'expression `local_ck` renvoie la valeur de l'horloge locale, que l'on peut trouver au sommet de la pile s.
- L'expression `run` e_p évalue e_p en une définition de processus puis l'exécute.
- L'expression `let/and/in` exécute en parallèle e_1 et e_2. Le choix que l'on a fait pour les statuts des expressions permet d'exprimer très simplement la règle du produit synchrone. Il suffit de regarder le minimum des statuts des deux branches. Si les deux branches ont terminé leur exécution, alors ce minimum est 0 et on doit exécuter l'expression e_3 (règle LET-DONE). Sinon, le minimum des deux statuts est l'horloge la plus rapide qu'attendent les deux branches du parallèle. C'est cette horloge que doit attendre la composition parallèle des deux expressions (règle LETPAR).
- La définition d'un signal (`signal/in`) évalue dans un premier temps tous ses paramètres, puis ajoute un nouveau signal dans l'environnement de signaux, associé à un nom frais n. Ce nouveau signal a une valeur par défaut v_d, une fonction de combinaison v_g, sa dernière valeur est initialisée à v_d, m est l'ensemble des valeurs émises, b indique s'il s'agit d'un signal à mémoire et ck'' est son horloge de réinitialisation. On exécute ensuite le corps e_1 en remplaçant la variable x par le nom du signal. On initialise le multi-ensemble des valeurs émises sur le signal par un multi-ensemble m contenant toutes les valeurs émises pendant l'instant courant. Il faut en quelque sorte deviner ces valeurs au moment de la création du signal. Nous verrons dans la partie 4.5 que cela cache en fait un calcul de point fixe.
- L'émission d'un signal par un appel à `emit` consiste à associer la valeur v au nom n dans l'événement $\mathcal{E}$. Plus précisément, il faut l'ajouter dans l'événement local associé à l'instant courant de l'horloge du signal, qui est désigné par la pile $s_{|ck'}$.
- De la même façon, l'opérateur `last` accède à la dernière valeur du signal qui est stockée dans l'environnement local de signaux correspondant à l'instant courant de son horloge.
- On choisit d'être un peu plus restrictif que la sémantique originale de REACTIVEML pour la séparation entre expressions instantanées et expressions réactives, afin de simplifier l'écriture de la sémantique. En particulier, on interdit l'émission de signaux dans de nombreuses expressions, ce que l'on peut voir dans les règles par le fait que l'on force l'expression à

$$\emptyset, s \vdash v \xrightarrow[\mathcal{S}]{\{\},0} v \qquad \emptyset, s \vdash \texttt{local\_ck} \xrightarrow[\mathcal{S}]{\{\},0} \texttt{top}(s) \qquad \frac{\emptyset, s \vdash e_1 \xrightarrow[\mathcal{S}]{\{\},0} v_1 \qquad \emptyset, s \vdash e_2 \xrightarrow[\mathcal{S}]{\{\},0} v_2}{\emptyset, s \vdash (e_1, e_2) \xrightarrow[\mathcal{S}]{\{\},0} (v_1, v_2)}$$

$$\frac{\emptyset, s \vdash e_1 \xrightarrow[\mathcal{S}]{\{\},0} \lambda x.\ e_1' \qquad \emptyset, s \vdash e_2 \xrightarrow[\mathcal{S}]{\{\},0} v_2 \qquad N, s \vdash e_1'[x \leftarrow v_2] \xrightarrow[\mathcal{S}]{\mathcal{E},0} v}{N, s \vdash e_1\ e_2 \xrightarrow[\mathcal{S}]{\mathcal{E},0} v}$$

$$\frac{N, s \vdash e_1[x \leftarrow \texttt{rec}\ x = e_1] \xrightarrow[\mathcal{S}]{\mathcal{E},0} v}{N, s \vdash \texttt{rec}\ x = e_1 \xrightarrow[\mathcal{S}]{\mathcal{E},0} v} \qquad \frac{\emptyset, s \vdash e_p \xrightarrow[\mathcal{S}]{\{\},0} \texttt{process}\ e_1 \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'}{N_1, s \vdash \texttt{run}\ e_p \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'}$$

$$\text{(LETPAR)}\ \frac{N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,k_1} e_1' \qquad N_2, s \vdash e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_2,k_2} e_2' \qquad k = \min_s(k_1, k_2) \neq 0}{N_1 \cdot N_2, s \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3 \xrightarrow[\mathcal{S}]{\mathcal{E}_1 \sqcup \mathcal{E}_2, k} \texttt{let}\ x_1 = e_1'\ \texttt{and}\ x_2 = e_2'\ \texttt{in}\ e_3}$$

$$\text{(LETDONE)}\ \frac{N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,0} v_1 \qquad N_2, s \vdash e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_2,0} v_2 \qquad N_3, s \vdash e_3[x_1 \leftarrow v_1; x_2 \leftarrow v_2] \xrightarrow[\mathcal{S}]{\mathcal{E}_3,k} e_3'}{N_1 \cdot N_2 \cdot N_3, s \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3 \xrightarrow[\mathcal{S}]{\mathcal{E}_1 \sqcup \mathcal{E}_2 \sqcup \mathcal{E}_3, k} e_3'}$$

$$\frac{\begin{array}{c}\emptyset, s \vdash e_{ck} \xrightarrow[\mathcal{S}]{\{\},0} ck' \qquad \emptyset, s \vdash e_d \xrightarrow[\mathcal{S}]{\{\},0} v_d \qquad \emptyset, s \vdash e_g \xrightarrow[\mathcal{S}]{\{\},0} v_g \\ \emptyset, s \vdash e_{rck} \xrightarrow[\mathcal{S}]{\{\},0} ck'' \qquad N, s \vdash e_1[x \leftarrow n^{ck'}] \xrightarrow[\mathcal{S}]{\mathcal{E},k} e' \qquad \mathcal{S}(s_{|ck'})(n) = (v_d, v_g, v_d, m, b, ck'')\end{array}}{N \cdot \{n\}, s \vdash \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e_{ck}, \texttt{d} : e_d, \texttt{g} : e_g, \texttt{rck} : e_{rck})\ \texttt{in}\ e_1 \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'}$$

$$\frac{\begin{array}{c}\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad \emptyset, s \vdash e_1 \xrightarrow[\mathcal{S}]{\{\},0} v \\ \mathcal{E} = \{s_{|ck'} \mapsto \{n \mapsto \{v\}\}\}\end{array}}{\emptyset, s \vdash \texttt{emit}\ e_s\ e_1 \xrightarrow[\mathcal{S}]{\mathcal{E},0} ()} \qquad \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'}}{\emptyset, s \vdash \texttt{last}\ e_s \xrightarrow[\mathcal{S}]{\{\},0} \mathcal{S}^l(s_{|ck'})(n)}$$

$$\frac{\emptyset, s \vdash e_{ck} \xrightarrow[\mathcal{S}]{\{\},0} ck' \qquad \texttt{eoi}_{s,\mathcal{S}}(ck')}{\emptyset, s \vdash \texttt{pause}\ e_{ck} \xrightarrow[\mathcal{S}]{\{\},ck'} ()} \qquad \frac{\emptyset, s \vdash e_{ck} \xrightarrow[\mathcal{S}]{\{\},0} ck' \qquad not(\texttt{eoi}_{s,\mathcal{S}}(ck'))}{\emptyset, s \vdash \texttt{pause}\ e_{ck} \xrightarrow[\mathcal{S}]{\{\},ck'} \texttt{pause}\ ck'}$$

$$\frac{\emptyset, s \vdash e_{ck} \xrightarrow[\mathcal{S}]{\{\},0} ck' \qquad \texttt{eoi}_{s,\mathcal{S}}(ck')}{\emptyset, s \vdash \texttt{qpause}\ e_{ck} \xrightarrow[\mathcal{S}]{\{\},\top_{ck}} ()} \qquad \frac{\emptyset, s \vdash e_{ck} \xrightarrow[\mathcal{S}]{\{\},0} ck' \qquad not(\texttt{eoi}_{s,\mathcal{S}}(ck'))}{\emptyset, s \vdash \texttt{qpause}\ e_{ck} \xrightarrow[\mathcal{S}]{\{\},\top_{ck}} \texttt{qpause}\ ck'}$$

FIGURE 4.2 – Sémantique comportementale (1)

$$\text{(PRESENTPRES)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck} \qquad n \in \mathcal{S}(s) \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,k} e_1'}{N_1, s \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,k} e_1'}$$

$$\text{(PRESENTABS)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck} \qquad n \notin \mathcal{S}(s)}{\emptyset, s \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[\mathcal{S}]{\{\},ck} e_2}$$

$$\text{(UNTILDONE)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,0} v}{N_1, s \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,0} v}$$

$$\text{(UNTILABS)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,ck''} e_1' \qquad n \notin \mathcal{S}(s_{\restriction ck'})}{N_1, s \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,ck''} \texttt{do}\ e_1'\ \texttt{until}\ n^{ck'}(x) \to e_2}$$

$$\text{(UNTILPRESEOI)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,ck''} e_1' \qquad n \in \mathcal{S}(s_{\restriction ck'}) \qquad \texttt{eoi}_{s,\mathcal{S}}(ck')}{N_1, s \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,ck'} e_2[x \leftarrow \mathcal{S}^v(s_{\restriction ck'})(n)]}$$

$$\text{(UNTILPRESNOTEOI)}\ \frac{\begin{array}{c}\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,ck''} e_1' \qquad n \in \mathcal{S}(s_{\restriction ck'}) \\ not(\texttt{eoi}_{s,\mathcal{S}}(ck')) \qquad k = \min_s(ck', ck'')\end{array}}{N_1, s \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,k} \texttt{do}\ e_1'\ \texttt{until}\ n^{ck'}(x) \to e_2}$$

$$\text{(WHENDONE)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck} \qquad n \in \mathcal{S}(s) \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,0} v}{N_1, s \vdash \texttt{do}\ e_1\ \texttt{when}\ e_s \xrightarrow[\mathcal{S}]{\mathcal{E}_1,0} v}$$

$$\text{(WHENPRES)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck} \qquad n \in \mathcal{S}(s) \qquad N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,k} e_1'}{N_1, s \vdash \texttt{do}\ e_1\ \texttt{when}\ e_s \xrightarrow[\mathcal{S}]{\mathcal{E}_1,k} \texttt{do}\ e_1'\ \texttt{when}\ n^{ck}}$$

$$\text{(WHENABS)}\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck} \qquad n \notin \mathcal{S}(s)}{\emptyset, s \vdash \texttt{do}\ e_1\ \texttt{when}\ e_s \xrightarrow[\mathcal{S}]{\{\},\top_{ck}} \texttt{do}\ e_1\ \texttt{when}\ n^{ck}}$$

FIGURE 4.3 – Sémantique comportementale (2) : structures de contrôle

émettre l'événement vide $\{\}$. On limite de la même façon la création de signal, ce qui fait que l'on peut également supposer que l'ensemble des noms créés par ces expressions est vide.
– La gestion de l'accès aux signaux se fait grâce à l'opération de restriction de la pile courante s. Un signal $n^{ck'}$ peut être lu si son horloge ck' est plus lente que l'horloge locale, c'est-à-dire si son horloge apparait dans la pile courante. Si ce n'est pas le cas, alors la pile $s_{\restriction ck'}$ n'est pas définie et l'expression n'a donc pas de sémantique. Si $s_{\restriction ck'}$ est bien définie, alors $\mathcal{S}(s_{\restriction ck'})$ est l'environnement local de signaux correspondant à l'instant actuel de l'horloge ck', où l'on doit lire la valeur du signal $n^{ck'}$. Cette sémantique explique pourquoi on ne peut pas accéder à un signal en dehors du domaine réactif auquel il est attaché. On ne sait en effet pas quel environnement local de signaux utiliser, puisque la pile ne nous donne pas l'indice de l'instant courant de cette horloge.

On peut également voir la sémantique de `pause` et `qpause` :
– Le cas de `pause` permet de voir l'utilité du statut k dans la règle de sémantique. Une expression de statut 0 termine instantanément, alors qu'une expression de statut ck' attend un prochain instant. Afin que `pause` ck' attende bien le prochain instant de l'horloge ck', elle ne se réécrit que lorsque le prédicat $\texttt{eoi}_{s,\mathcal{S}}(ck')$ est vérifié. En effet, dans ce cas, le prochain instant de l'horloge locale aura bien lieu dans l'instant suivant de l'horloge ck', pendant lequel l'expression doit terminer. On peut remarquer au passage que puisque $\texttt{eoi}_{s,\mathcal{S}}(ck')$ n'est pas défini pour les horloges qui n'apparaissent pas dans s, `pause` ck' n'a pas de sémantique si ck' est plus rapide que l'horloge locale.
– L'expression `qpause` ck' permet également d'attendre l'instant suivant de l'horloge ck', mais elle doit être considérée en attente par le domaine réactif. C'est pourquoi le statut de retour de cette expression est $\top_{ck}$, qui est l'horloge la plus lente du programme. Si on met cette expression en parallèle avec une autre expression e_2, alors le statut de retour de l'expression complète sera celui de e_2, puisque l'on prend le minimum des statuts, dont $\top_{ck}$ est le maximum. On obtient donc bien le comportement voulu : l'expression `qpause` ck' n'influence pas le choix de l'attente du domaine réactif. Un cas limite est lorsque le domaine ne contient que des processus utilisant `qpause`. Dans ce cas, le domaine attend le prochain instant de l'horloge globale.

Structures de contrôle La figure 4.3 montre les règles concernant les structures de contrôle :
– Le test de présence d'un signal, avec la construction `present`, exécute e_1 immédiatement si le signal n^{ck} est présent (PRESENTPRES). Sinon, il exécute e_2 à l'instant suivant, puisque son statut est égal à l'horloge locale ck (PRESENTABS). Ce test constitue une dépendance instantanée. Il ne peut donc se faire que pour un signal sur l'horloge locale. Cela se traduit dans les règles par le fait que l'on regarde la présence du signal dans l'environnement local de signaux $\mathcal{S}(s)$ correspondant à la pile courante, qui contient les valeurs des signaux sur l'horloge locale $\texttt{top}(s)$.
– En REACTIVEML, la préemption est faible. Cela signifie que la préemption d'un processus n'a lieu qu'à la fin de l'instant si le signal est présent au cours de cet instant. Dans le langage étendu avec les domaines réactifs, la préemption a lieu à la fin de l'instant de l'horloge du signal. Tant que le signal est absent, la construction `do`/`until` se comporte comme son corps (UNTILABS). Si le corps termine, alors le `do`/`until` termine et renvoie la même valeur (UNTILDONE). Lorsque le signal est présent, on attend d'être dans le dernier instant de l'horloge locale inclus dans l'instant courant de l'horloge ck' du signal pour faire la préemption (UNTILPRESEOI). Tant que ce n'est pas le cas, on renvoie un statut égal au minimum de l'horloge du signal et du statut renvoyé par le corps (UNTILPRESNOTEOI). Cela signifie que l'on exécute le corps tant qu'il fait des instants sur une horloge plus rapide que celle du signal.
– La construction `do`/`when` exécute son corps e_1 uniquement aux instants où le signal n^{ck} est présent (WHENPRES). Lorsque le corps termine, elle termine instantanément en renvoyant la même valeur (WHENDONE). Enfin, lorsque le signal est absent, le corps n'est pas exécuté et l'expression renvoie le statut $\top_{ck}$ qui indique qu'elle est bloquée en attente d'un signal (WHENABS).

$$\text{(Inst)}\ \frac{\emptyset, s \vdash e_b \xrightarrow[\mathcal{S}]{\{\},0} j \qquad N, s \vdash (e_1[x \leftarrow ck']) \texttt{ in } (ck', 0/j) \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'}{N \cdot \{ck'\}, s \vdash \texttt{domain } x \texttt{ by } e_b \texttt{ do } e_1 \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'}$$

$$\text{(LocalEoi)}\ \frac{i < j - 1 \qquad N_1, s :: (ck', i) \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1, ck'} e_1' \qquad N, s \vdash e_1' \texttt{ in } (ck', i+1/j) \xrightarrow[\mathcal{S}]{\mathcal{E},k} e''}{N_1 \cdot N, s \vdash e_1 \texttt{ in } (ck', i/j) \xrightarrow[\mathcal{S}]{\mathcal{E}_1 \sqcup \mathcal{E},k} e''}$$

$$\text{(Term)}\ \frac{i < j \qquad N, s :: (ck', i) \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E},0} v}{N, s \vdash e_1 \texttt{ in } (ck', i/j) \xrightarrow[\mathcal{S}]{\mathcal{E},0} v}$$

$$\text{(CounterEoi)}\ \frac{N, s :: (ck', j-1) \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}, ck''} e_1' \qquad k = \max_s(ck'', \texttt{top}(s))}{N, s \vdash e_1 \texttt{ in } (ck', j-1/j) \xrightarrow[\mathcal{S}]{\mathcal{E},k} e_1' \texttt{ in } (ck', 0/j)}$$

$$\text{(ParentEoi)}\ \frac{i < j - 1 \qquad N, s :: (ck', i) \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}, ck''} e_1' \qquad ck' \preceq_s ck''}{N, s \vdash e_1 \texttt{ in } (ck', i/j) \xrightarrow[\mathcal{S}]{\mathcal{E}, ck''} e_1' \texttt{ in } (ck', 0/j)}$$

FIGURE 4.4 – Sémantique comportementale (3) : domaines réactifs

Domaines réactifs La dernière figure 4.4 montre les règles associées au fonctionnement des domaines réactifs :

- On initialise tout d'abord le domaine en évaluant la borne e_b donnée après le `by`, puis en créant un nom frais d'horloge ck' que l'on substitue à la variable x dans le corps du domaine (règle Inst). On obtient alors un domaine en cours d'exécution noté e_1 `in` $(ck', i/j)$. On rappelle que ck' est l'horloge du domaine réactif, i est le nombre d'instants effectués dans l'instant courant de l'horloge parente et j est le nombre maximal d'instants par instant de l'horloge parente.
- On exécute ensuite des instants locaux (règle LocalEoi). Pour cela, on exécute le corps e_1 du domaine réactif en ajoutant l'horloge locale ck' sur la pile, avec l'indice i de l'instant à exécuter. On incrémente ensuite le compteur du domaine, puis on effectue le pas suivant.
- Si le corps termine son exécution et se réécrit en une valeur v, alors le domaine termine également instantanément et renvoie la même valeur (règle Term).
- Sinon, on peut avoir deux cas de figure pour arrêter l'exécution de ces instants locaux :
  - Soit le compteur d'instants atteint la valeur maximale $j - 1$ (règle CounterEoi). Si son corps attend le prochain instant de l'horloge locale, alors le domaine attend le prochain instant de son horloge parente. Sinon, il attend la même horloge que son corps.
  - Soit le corps du domaine est bloqué en attente d'une horloge plus lente, c'est-à-dire que son code de retour est une horloge ck'' plus lente que ck' (règle ParentEoi). Le domaine réactif renvoie alors le même statut que son corps et remet à zéro son compteur d'instants. Cette règle implémente l'attente automatique que l'on a décrite dans la partie 3.1.

Propriétés

Nous allons maintenant énoncer les deux propriétés majeures de la sémantique comportementale qui sont conservées dans notre extension du langage : le *déterminisme* et l'*unicité* de la réaction. Le déterminisme signifie que dans un environnement de signaux donné, un programme ne peut réagir que d'une seule façon.

Propriété 4.1 (Déterminisme)**.** *La sémantique comportementale de toute expression e est déterministe, c'est-à-dire que si toutes les fonctions de combinaison des signaux sont associatives et commutatives, et si $N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E}_1, k_1} e_1$ et $N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E}_2, k_2} e_2$, alors $\mathcal{E}_1 = \mathcal{E}_2$, $k_1 = k_2$ et $e_1 = e_2$.*

Démonstration. La preuve se fait par induction sur les dérivations. Il faut vérifier que, si plusieurs règles sont possibles pour une expression (par exemple LETPAR et LETDONE), alors les conditions supplémentaires ne peuvent être vérifiées que pour une seule règle à la fois. Les cas les plus intéressants sont les suivants :

Cas $\texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2$**.** Comme pour la preuve de déterminisme de REACTIVEML [MP08b], on utilise le fait que la valeur du signal est déterministe puisque toutes les fonctions de combinaison sont associatives et commutatives.

Cas $e_1\ \texttt{in}\ (ck', i/j)$**.** On peut remarquer que toutes les règles font réagir le corps e_1 du domaine dans la pile $s' = s :: (ck', i)$. On peut donc appliquer l'hypothèse d'induction sur cette expression. On conclut ensuite en vérifiant que les différentes règles pour les domaines réactifs ont des conditions disjointes. □

La seconde propriété qui nous intéresse est l'unicité. Elle exprime le fait que si une expression est réactive, c'est-à-dire s'il existe un environnement de signaux dans lequel elle peut réagir, alors il existe un plus petit environnement de signaux dans lequel elle peut réagir.

Propriété 4.2 (Unicité)**.** *Pour tout expression e et pile s, l'ensemble $S = \{\mathcal{S} \mid \exists N, \mathcal{E}, k.\ N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E}, k} e'\}$ des environnements de signaux dans lequel e peut réagir admet un plus petit élément $\sqcap S$.*

Le principe de la preuve est le même que pour celle de REACTIVEML [MP08b]. Elle s'appuie sur le lemme suivant, qui montre que si une expression peut réagir dans deux environnements de signaux, elle peut réagir dans l'intersection de ces deux environnements. Ce lemme repose en particulier sur le délai de la réaction à l'absence.

Lemme 4.3. *Supposons $N, s \vdash e \xrightarrow[\mathcal{S}_1]{\mathcal{E}_1, k_1} e_1$ et $N, s \vdash e \xrightarrow[\mathcal{S}_2]{\mathcal{E}_2, k_2} e_2$. Soit $\mathcal{S}_3 = \mathcal{S}_1 \sqcap \mathcal{S}_2$. Alors il existe $\mathcal{E}_3$, k_3 et e_3 tels que $N, s \vdash e \xrightarrow[\mathcal{S}_3]{\mathcal{E}_3, k_3} e_3$.*

Démonstration. Par induction sur la dérivation. □

4.4 Exemples

Sémantique des expressions dérivées

Un premier exercice pour manipuler les règles de la sémantique comportementale est de donner les règles des expressions dérivées du noyau. Nous allons donner des versions simplifiées de ces règles, équivalentes aux règles données précédemment. L'équivalence signifie ici que dans le même environnement de signaux, l'expression dérivée et son encodage renvoient le même statut et émettent les mêmes signaux.

On peut commencer par le cas de la séquence, qui sera utile pour les exemples suivants. On rappelle qu'elle s'encode par :

$$e_1; e_2 \triangleq \texttt{let \_} = e_1\ \texttt{and \_} = ()\ \texttt{in}\ e_2$$

On déduit les réductions suivantes de cet encodage :

$$\textsc{LetPar}\ \frac{N, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E},ck} e_1' \qquad \emptyset, s \vdash () \xrightarrow[\mathcal{S}]{\{\},0} () \qquad \min_s(ck, 0) = ck}{\dfrac{N, s \vdash \texttt{let \_} = e_1 \texttt{ and \_} = () \texttt{ in } e_2 \xrightarrow[\mathcal{S}]{\mathcal{E},ck} \texttt{let \_} = e_1' \texttt{ and \_} = () \texttt{ in } e_2}{N, s \vdash e_1; e_2 \xrightarrow[\mathcal{S}]{\mathcal{E},ck} e_1'; e_2}}$$

$$\textsc{LetDone}\ \frac{N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,0} v \qquad \emptyset, s \vdash () \xrightarrow[\mathcal{S}]{\{\},0} () \qquad N_2, s \vdash e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_2,k} e_2'}{\dfrac{N_1 \cdot N_2, s \vdash \texttt{let \_} = e_1 \texttt{ and \_} = () \texttt{ in } e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_1 \sqcup \mathcal{E}_2,k} e_2'}{N_1 \cdot N_2, s \vdash e_1; e_2 \xrightarrow[\mathcal{S}]{\mathcal{E}_1 \sqcup \mathcal{E}_2,k} e_2'}}$$

Dans la première dérivation, l'expression e_1 se réécrit en e_1' et renvoie le statut ck, ce qui signifie qu'elle n'a pas terminé son exécution et qu'elle attend l'instant suivant de ck. La séquence renvoie alors le même statut. Dans la second dérivation, e_1 termine donc on exécute e_2 qui se réécrit en e_2'. La séquence se réécrit alors en e_2' et renvoie le statut k de e_2. Une autre construction intéressante est l'attente de la valeur d'un signal :

$$\texttt{await } e_s(x) \texttt{ in } e_1 \triangleq \texttt{do (loop (qpause local\_ck)) until } e_s(x) \to e_1$$

La particularité de cette construction est qu'elle utilise l'opérateur `qpause` pour ne pas rendre le domaine réactif où elle est utilisée non coopératif. On obtient les règles suivantes pour sa sémantique :

$$(\textsc{AwaitAbs})\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad n \notin \mathcal{S}(s)}{N, s \vdash \texttt{await } e_s(x) \texttt{ in } e_1 \xrightarrow[\mathcal{S}]{\{\},\top_{ck}} \texttt{await } n^{ck'}(x) \texttt{ in } e_1}$$

$$(\textsc{AwaitPresEoi})\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad n \in \mathcal{S}(s) \qquad \texttt{eoi}_{s,\mathcal{S}}(ck')}{N, s \vdash \texttt{await } e_s(x) \texttt{ in } e_1 \xrightarrow[\mathcal{S}]{\{\},ck'} e_1[x \leftarrow \mathcal{S}^v(s_{|ck'})(n)]}$$

$$(\textsc{AwaitPresNotEoi})\ \frac{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\{\},0} n^{ck'} \qquad n \in \mathcal{S}(s) \qquad not(\texttt{eoi}_{s,\mathcal{S}}(ck'))}{N, s \vdash \texttt{await } e_s(x) \texttt{ in } e_1 \xrightarrow[\mathcal{S}]{\{\},ck'} \texttt{await } n^{ck'}(x) \texttt{ in } e_1}$$

Tant que le signal est absent, la construction `await` est en attente et renvoie comme statut l'horloge la plus lente du programme, c'est-à-dire $\top_{ck}$ (AwaitAbs). Cela correspond à la règle UntilAbs, puisque le corps du `do`/`until`, c'est-à-dire `loop` (`qpause local_ck`), renvoie toujours le statut $\top_{ck}$. Lorsque le signal est présent, l'expression ne se réécrit que lorsque l'on est dans le dernier instant correspondant à l'horloge du signal (AwaitPresEoi). On retrouve l'équivalent des règles UntilPresEoi et UntilPresNotEoi. Dans la dernière règle, on renvoie le statut ck' car on a $k = \min_s(ck', \top_{ck}) = ck'$ dans la règle UntilPresNotEoi.

Un premier exemple de domaine réactif

Nous allons maintenant considérer un exemple simple afin de rendre les nombreuses notations de la partie 4.2 un peu plus concrètes et d'illustrer les règles de la sémantique :

```
let process simple_domain =
  signal s1 clock global_ck default 0 gather (+) in
  domain ck do
    memory s2 default 0 gather (+) in
    emit s2 4;
    pause ck;
    emit s2 1;
    pause global_ck;
    emit s1 (last s2)
  done
```

On définit un signal `s1` sur l'horloge globale et un signal `s2` sur l'horloge `ck`. On note n_1 et n_2 les noms de ces signaux. Au cours du premier instant de l'horloge globale, on émet `4` sur le signal `s2` pendant le premier instant de `ck` et `1` pendant le second, puis on attend l'instant suivant de l'horloge globale. On émet ensuite sur `s1` la valeur précédente de `s2`, qui est `5 = 0 + 4 + 1` puisqu'il s'agit d'un signal à mémoire.

Pour simplifier les dérivations, nous allons omettre tout ce qui concerne l'instanciation des signaux et du domaine réactif. On omet également les horloges sur les noms de signaux. On note $s_i \triangleq (\top_{ck}, i)$. On s'intéresse donc à la sémantique de l'expression e_0 `in` $(ck, 0/\infty)$ avec :

$$\begin{aligned} e_0 &\triangleq \texttt{emit}\ n_2\ 4; (\texttt{pause}\ ck; e_1) \\ e_1 &\triangleq \texttt{emit}\ n_2\ 1; (\texttt{pause}\ \top_{ck}; e_2) \\ e_2 &\triangleq \texttt{emit}\ n_1\ (\texttt{last}\ n_2) \end{aligned}$$

Avant de pouvoir calculer cette dérivation, il faut tout d'abord connaître l'environnement de signaux $\mathcal{S}_0$ dans lequel l'expression doit réagir. A partir de la sémantique intuitive que l'on a donnée au chapitre 3, on peut deviner l'environnement suivant :

$$\begin{aligned} \mathcal{S}_0 \triangleq \{&(\top_{ck}, 0) \mapsto \{n_1 \mapsto (0, +, 0, \emptyset, \mathit{ff}, \bot_{ck})\}; \\ &(\top_{ck}, 0) :: (ck, 0) \mapsto \{n_2 \mapsto (0, +, 0, \{4\}, \mathit{tt}, \bot_{ck})\}; \\ &(\top_{ck}, 0) :: (ck, 1) \mapsto \{n_2 \mapsto (0, +, 4, \{1\}, \mathit{tt}, \bot_{ck})\}\} \end{aligned}$$

Le signal n_1 n'est pas émis au cours du premier instant de l'horloge globale. Le signal n_2 est émis au cours des deux instants de ck et il s'agit d'un signal à mémoire qui n'est jamais réinitialisé. On peut vérifier que cet environnement est bien formé. Le successeur de $(\top_{ck}, 0) :: (ck, 0)$ est $\texttt{succ}((\top_{ck}, 0) :: (ck, 0), \mathit{Dom}(\mathcal{S}_0)) = (\top_{ck}, 0) :: (ck, 1)$. On vérifie que les deux environnements locaux de signaux correspondants donnent les mêmes valeurs par défaut, fonctions de combinaison et horloges de réinitialisation. La valeur précédente au cours du second instant de ck est aussi bien celle émise au premier instant.

On calcule tout d'abord la dérivation associée au premier instant de l'horloge ck, inclus dans le premier instant de l'horloge $\top_{ck}$. Cela correspond à l'émission de la valeur `4` sur `s_2`. On obtient la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\mathcal{E}_0 \triangleq \{s_0 :: (ck, 0) \mapsto \{n_2 \mapsto \{4\}\}\}}{\emptyset, s_0 :: (ck, 0) \vdash \texttt{emit}\ n_2\ 4 \xrightarrow[\mathcal{S}_0]{\mathcal{E}_0, 0} ()} \quad \dfrac{\dfrac{\texttt{eoi}_{s_0 :: (ck,0), \mathcal{S}_0}(ck)}{\emptyset, s_0 :: (ck, 0) \vdash \texttt{pause}\ ck \xrightarrow[\mathcal{S}_0]{\{\}, ck} ()}}{\emptyset, s_0 :: (ck, 0) \vdash \texttt{pause}\ ck; e_1 \xrightarrow[\mathcal{S}_0]{\{\}, ck} (); e_1}}{\emptyset, s_0 :: (ck, 0) \vdash \texttt{emit}\ n_2\ 4; (\texttt{pause}\ ck; e_1) \xrightarrow[\mathcal{S}_0]{\mathcal{E}_0, ck} (); e_1}}{\emptyset, s_0 :: (ck, 0) \vdash e_0 \xrightarrow[\mathcal{S}_0]{\mathcal{E}_0, ck} (); e_1}$$

On a bien $\mathcal{E}_0 \subseteq \mathcal{S}_0^m$ et $\texttt{eoi}_{s_0 :: (ck,0), \mathcal{S}_0}(ck)$ car :

$$(\texttt{succ}(s_0 :: (ck, 0), \mathit{Dom}(\mathcal{S}_0)))\,|_{ck} = s_0 :: (ck, 1) \neq s_0 :: (ck, 0) = (s_0 :: (ck, 0))\,|_{ck}$$

On peut faire de même avec le second instant de l'horloge ck, c'est-à-dire l'émission de `1` sur `s_2` (on omet la dérivation qui réécrit $();e_1$ en e_1 qui est évidente) :

$$\dfrac{\dfrac{\mathcal{E}_1 \triangleq \{s_0 :: (ck,1) \mapsto \{n_2 \mapsto \{1\}\}\}}{\emptyset, s_0 :: (ck,1) \vdash \texttt{emit}\ n_2\ 1 \xrightarrow[\mathcal{S}_0]{\mathcal{E}_1,0} ()} \quad \dfrac{\dfrac{s_0 :: (ck,1) = \texttt{last}(ck, Dom(\mathcal{S}_0))}{\texttt{eoi}_{s_0::(ck,1),\mathcal{S}_0}(\top_{ck})}}{\emptyset, s_0 :: (ck,1) \vdash \texttt{pause}\ \top_{ck} \xrightarrow[\mathcal{S}_0]{\{\},\top_{ck}} ()}}{\dfrac{\emptyset, s_0 :: (ck,1) \vdash \texttt{pause}\ \top_{ck}; e_2 \xrightarrow[\mathcal{S}_0]{\{\},\top_{ck}} (); e_2}{N, s_0 :: (ck,1) \vdash \texttt{emit}\ n_2\ 1; (\texttt{pause}\ \top_{ck}; e_2) \xrightarrow[\mathcal{S}_0]{\mathcal{E}_1,\top_{ck}} (); e_2}}{\emptyset, s_0 :: (ck,1) \vdash e_1 \xrightarrow[\mathcal{S}_0]{\mathcal{E}_1,\top_{ck}} (); e_2}$$

On remarque en particulier que l'expression `pause` $\top_{ck}$ se réécrit puisque le prochain instant de l'horloge ck est bien inclus dans le prochain instant de l'horloge globale.

On peut donc maintenant construire la dérivation correspondant au premier instant de l'horloge globale $\top_{ck}$:

$$\dfrac{\emptyset, s_0 :: (ck,0) \vdash e_0 \xrightarrow[\mathcal{S}_0]{\mathcal{E}_0,ck} (); e_1 \quad \dfrac{\emptyset, s_0 :: (ck,1) \vdash (); e_1 \xrightarrow[\mathcal{S}_0]{\mathcal{E}_1,\top_{ck}} (); e_2}{\emptyset, s_0 \vdash ((); e_1)\ \texttt{in}\ (ck, 1/\infty) \xrightarrow[\mathcal{S}_0]{\mathcal{E}_1,\top_{ck}} ((); e_2)\ \texttt{in}\ (ck, 0/\infty)}}{\emptyset, s_0 \vdash e_0\ \texttt{in}\ (ck, 0/\infty) \xrightarrow[\mathcal{S}_0]{\mathcal{E}_0 \sqcup \mathcal{E}_1,\top_{ck}} ((); e_2)\ \texttt{in}\ (ck, 0/\infty)}$$

La réaction du second instant du programme se fait dans l'environnement de signaux $\mathcal{S}_1$:

$$\begin{aligned}\mathcal{S}_1 \triangleq \{&(\top_{ck}, 1) \mapsto \{n_1 \mapsto (0, +, 0, \{10\}, \mathit{ff}, \bot_{ck});\\ &(\top_{ck}, 1) :: (ck, 0) \mapsto \{n_2 \mapsto (0, +, 5, \emptyset, \mathit{tt}, \bot_{ck})\}\}\end{aligned}$$

On peut vérifier que le signal n_2 est bien émis au cours du second instant de l'horloge globale et que sa valeur est 5. Le programme termine donc au second instant en émettant la valeur `5` sur `s_1`, puis en renvoyant la valeur $()$:

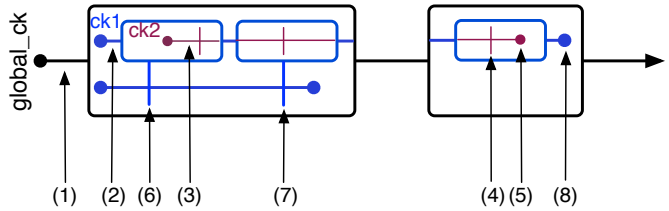
$$\dfrac{\emptyset, s_1 :: (ck,0) \vdash () \xrightarrow[\mathcal{S}_1]{\{\},0} () \quad \dfrac{\dfrac{\mathcal{S}^l(s_1 :: (ck,0)|_{ck})(n_2) = 5}{\emptyset, s_1 :: (ck,0) \vdash \texttt{last}\ n_2 \xrightarrow[\mathcal{S}_1]{\{\},0} 5} \quad \mathcal{E}_2 \triangleq \{s_1 \mapsto \{n_2 \mapsto \{5\}\}\}}{\emptyset, s_1 :: (ck,0) \vdash \texttt{emit}\ n_1\ (\texttt{last}\ n_2) \xrightarrow[\mathcal{S}_1]{\mathcal{E}_2,0} ()}}{\dfrac{\emptyset, s_1 :: (ck,0) \vdash (); \texttt{emit}\ n_1\ (\texttt{last}\ n_2) \xrightarrow[\mathcal{S}_1]{\mathcal{E}_2,0} ()}{\emptyset, s_1 \vdash ((); e_2)\ \texttt{in}\ (ck, 0/\infty) \xrightarrow[\mathcal{S}_1]{\mathcal{E}_2,0} ()}}$$

4.5 Discussion

Domaines réactifs en attente

Une des particularités importantes des domaines réactifs est qu'ils attendent automatiquement le prochain instant d'une horloge plus lente si jamais leur corps est bloqué en attente du prochain instant de cette horloge. La sémantique formelle montre que la situation est en fait un peu plus subtile. Considérons le processus suivant :

```
let process domain_waiting =
 (*1*)
 domain ck1 do
   (*2*)
  domain ck2 do
   (*3*) pause global_ck (*4*)
  done (*5*)
  ||
  (*6*) pause ck1 (*7*)
 done (*8*)
```



Sur le dessin, les barres verticales représentent les instants d'une horloge, alors que les traits horizontaux représentent des processus en parallèle. Les flèches indiquent les positions des points de contrôle mis en commentaire dans le code. Ainsi, au cours du premier instant de `ck1`, on effectue un seul instant de `ck2`, puisque son corps est bloqué en attente de `global_ck`. On effectue ensuite un second instant de `ck1`, puisque la seconde branche du parallèle le demande. La question est de savoir si on va ou non exécuter un instant de l'horloge `ck2` au cours de ce second instant de `ck1`. Nous avons fait le choix de répondre à cette question par l'affirmative. Dans l'exemple ci-dessus, l'exécution du second instant de `ck2` est invisible puisque le corps du domaine attend le prochain instant de l'horloge globale. Ce ne serait plus le cas si on utilisait la construction `qpause`. En supposant que l'on lance le processus `domain_waiting` dans l'horloge `global_ck`, l'ensemble $Dom(\mathcal{S})$ des piles décrivant les instants successifs du programme est :

$$\begin{aligned} Dom(\mathcal{S}) = \{&(\top_{ck}, 0), (\top_{ck}, 0) :: (ck_1, 0), (\top_{ck}, 0) :: (ck_1, 0) :: (ck_2, 0), \\ &(\top_{ck}, 0) :: (ck_1, 1), (\top_{ck}, 0) :: (ck_1, 1) :: (ck_2, 0), \\ &(\top_{ck}, 1), (\top_{ck}, 1) :: (ck_1, 0), (\top_{ck}, 1) :: (ck_1, 0) :: (ck_2, 0)\} \end{aligned}$$

Cela se traduit dans les règles par le fait qu'un domaine réactif effectue toujours au moins un instant local à chaque instant de son domaine parent. En effet, aucune règle ne permet de réécrire un domaine réactif sans d'abord évaluer son corps.

Nous avons fait ce choix car il est le plus simple à implémenter. Un domaine réactif ne fait alors que vérifier si son corps doit exécuter un autre instant local, comme le montrera la sémantique opérationnelle que nous présenterons au chapitre suivant. L'alternative aurait été de calculer l'horloge la plus rapide attendue par le corps, puis de se mettre en attente de cette horloge. Nous verrons au moment de présenter l'implémentation dans le chapitre 9 que cela n'est pas possible à cause de l'attente passive des signaux.

Sémantique des domaines réactifs

Nous aurions pu présenter la sémantique des domaines réactifs de façon un peu différente, afin de mettre en valeur le fait que l'exécution d'un domaine réactif au cours d'un instant de son domaine parent est indépendante des autres processus et domaines réactifs. Cela se traduit dans les règles par le fait que l'on ne regarde jamais la présence des signaux lents, mais seulement leur dernière valeur. On aurait alors pu écrire par exemple :

$$\frac{m < j - 1 \qquad \left(N_i, s :: (ck', i) \vdash e_i \xrightarrow[\mathcal{S}]{\mathcal{E}_i, ck'} e_{i+1}\right)_{i=0..m-1} \qquad N_m, s :: (ck', m) \vdash e_m \xrightarrow[\mathcal{S}]{\mathcal{E}_m, 0} v}{N_0 \cdot \ldots \cdot N_m, s \vdash e_0 \texttt{ in } (ck', 0/j) \xrightarrow[\mathcal{S}]{\mathcal{E}_0 \sqcup \cdots \sqcup \mathcal{E}_m, 0} v}$$

Cette règle correspond au cas où l'on exécute $m + 1$ instants de l'horloge locale pour obtenir une valeur au final. Pendant les m premiers instants, on réécrit le corps e_i du domaine en e_{i+1} avec un statut égal à l'horloge locale ck', ce qui implique que l'on doit exécuter un autre instant local. Enfin, pendant le dernier instant, le corps e_m se réécrit en une valeur v et termine. Le domaine réactif termine donc en renvoyant la même valeur. On obtient l'événement total en unissant les

événements $\mathcal{E}_i$ correspondant aux $m+1$ instants locaux. On aurait dû écrire une règle similaire pour les différentes combinaisons possibles des règles LocalEoi, CounterEoi et ParentEoi. On voit que cette présentation alourdit les règles par rapport au choix que nous avons fait.

Mais où est pre ?

Le problème du pre en ReactiveML classique Nous avons fait le choix dans la présentation de la syntaxe abstraite du langage d'omettre l'opérateur `pre`. Ce choix n'est pas dicté par des raisons stylistiques (par exemple alléger la sémantique), mais par des problèmes liés au comportement de cet opérateur.

L'opérateur `pre` permet de savoir si un signal a été émis à l'instant précédent, alors que l'opérateur `pre?` renvoie la valeur du signal à l'instant précédent s'il a été émis et sa valeur par défaut sinon. Ces deux opérateurs sont présents dans Esterel et ReactiveML. Par exemple, le processus suivant va afficher `10` :

exemples/ch4/pre_simple.rml

```
let process pre_simple =
  signal s default 0 gather (+) in
  emit s 10
  ||
  pause; print_int (pre? s)
> ./pre_when_ok.rml.native
10
```

L'opérateur `pre?` pose déjà un problème en ReactiveML classique dans son interaction avec la suspension. Supposons qu'on lance le processus précédent sous une suspension, avec un signal qui est présent lors des deux premiers instants :

exemples/ch4/pre_when_ok.rml

```
let process pre_when_ok =
  signal c in
  do run pre_simple when c done
  ||
  emit c; pause; emit c
```

Le programme affiche toujours la même valeur :

```
> ./pre_when_ok.rml.native
10
```

Supposons maintenant que le signal ne soit plus présent au second instant mais au troisième :

exemples/ch4/pre_when_bad.rml

```
let process pre_when_bad =
  signal c in
  do run pre_simple when c done
  ||
  emit c; pause; pause; emit c
> ./pre_when_bad.rml.native
0
```

On voit que le résultat de l'exécution de `pre_simple` a changé selon le contexte où il est lancé. Cela s'explique par le fait qu'en ReactiveML classique, tous les signaux ont la même horloge, qui est l'horloge globale du programme. L'opérateur `pre?` se réfère donc à la valeur à l'instant précédent de l'horloge globale, qui n'est pas forcément l'instant précédent où le processus a été exécuté en cas de suspension. Dans le cas de `pre_when_bad`, l'appel de `pre?` renvoie la valeur au deuxième instant du programme, alors que le signal a été émis au premier.

Une solution avec les domaines réactifs En permettant de créer des horloges locales, les domaines réactifs permettent de résoudre le problème précédent. Il suffit en effet de déclarer un domaine réactif local pour le corps du `do/when` :

exemples/ch4/pre_when_domain.rml

```
let process pre_when_domain =
  signal c in
  do
    domain ck by 1 do run pre_simple done
  when c done
  ||
  emit c; pause; pause; emit c
> ./pre_when_domain.rml.native
10
```

Le comportement du processus est alors le même quel que soit son contexte d'activation, puisque le signal `s` est défini dans le domaine réactif. L'opérateur **pre**? se réfère donc à l'instant précédent de l'horloge `ck`, qui est bien l'instant précédent où le corps du processus a été exécuté. Le domaine réactif empêche également au signal `s` d'échapper du domaine, ce qui est normal puisque son rythme d'activation est plus lent que l'horloge globale.

Un nouveau problème L'opérateur **pre** pose un autre problème dans le langage étendu avec les domaines réactifs. Considérons le programme suivant :

exemples/ch4/domain_pre.rml

```
let process domain_pre =
  domain ck by 10 do
    signal s default 0 gather (+) in
    emit s 10
    ||
    pause topck; print_int (pre? s)
  done
```

A la fin du premier instant de l'horloge `ck`, le corps du domaine attend le prochain instant de l'horloge `global_ck`, et donc le domaine attend automatiquement le prochain instant de son horloge parente, sans faire les dix instants locaux. Le programme affiche donc :

```
> ./domain_pre.rml.native
10
```

La première remarque que l'on peut faire est que la possibilité de tester la présence du signal à l'instant précédent fait que l'attente automatique des domaines réactifs n'est plus transparente. Le résultat du programme change selon que l'on attende automatiquement l'instant suivant, auquel cas il affiche `10`, ou que l'on fasse réellement 10 instants locaux, où le résultat deviendrait `0`.

L'utilisation de l'opérateur **pre**? reste fragile dans le cas des domaines réactifs, puisque le résultat du programme peut changer si l'on ajoute un nouveau domaine réactif parent comme dans l'exemple suivant :

exemples/ch4/domain_pre_bad.rml

```
let process domain_pre_bad =
  domain ck1 do
    run domain_pre
    ||
    pause ck1; pause ck1
  done

> ./domain_pre_bad.rml.native
0
```

En effet, comme on l'a vu au début de cette partie, le domaine réactif d'horloge `ck` à l'intérieur du processus `domain_pre` va effectuer un instant au cours du second instant de l'horloge `ck1`. Ce n'est qu'à l'instant suivant de l'horloge `ck` que l'on va utiliser l'opérateur **pre**? qui ne renverra donc pas la valeur attendue.

Face à toutes les subtilités liées à l'utilisation du **pre**?, nous avons décidé de le considérer comme une extension du langage pour les utilisateurs experts. C'est pourquoi nous ne l'avons pas inclus dans notre noyau du langage, bien que l'on puisse sans difficulté donner sa sémantique, en suivant ce qui a été fait en REACTIVEML classique dans [Man06]. Dans la plupart des utilisations, l'opérateur **last** permet d'obtenir le même résultat sans tous ces problèmes. On peut remarquer que dans tous nos exemples, nous n'avons jamais eu besoin de **pre**?.

Calcul de l'environnement de signaux

La sémantique comportementale, introduite pour la première fois pour ESTEREL [BG92], ne fait que décrire ce qu'est une bonne réaction synchrone sans se préoccuper de comment l'obtenir. Elle semble un peu magique : il faut deviner l'environnement de signaux minimal dans lequel doit réagir l'expression avant même de calculer sa dérivation. Dans le langage étendu avec les domaines, les choses sont encore plus surprenantes puisque l'on connait déjà à l'avance le nombre d'instants que doit faire chaque domaine réactif, car il est donné par le domaine de l'environnement de signaux. On a également décidé à l'avance du moment où le domaine réactif doit attendre son domaine parent. Tout ceci peut s'expliquer par le fait que l'on doit en fait calculer un point fixe. On ne doit jamais faire d'hypothèse mais simplement propager les informations acquises sur la présence et la valeur des signaux, ainsi que le statut des domaines réactifs. La sémantique opérationnelle du chapitre suivant montrera comment se passer de point fixe et exécuter un programme de façon incrémentale et efficace. Nous allons maintenant expliquer le calcul du point fixe, d'abord dans le cas de REACTIVEML classique, puis dans le langage étendu.

Cas de REACTIVEML Dans le cas de REACTIVEML, on suppose au départ que tous les signaux sont absents. On calcule alors la dérivation de la sémantique qui va donner de nouvelles informations sur la présence des signaux. On utilise ensuite cette information pour calculer une nouvelle dérivation prenant en compte les signaux présents. On itère ce processus jusqu'à ce que l'on atteigne un point fixe, c'est-à-dire que l'ensemble des signaux émis soit égal à l'ensemble des signaux présents.

Plus formellement, on calcule le point fixe d'une fonction notée $[\![e]\!]$ comme dans [Ber96]. Cette fonction est définie par :

$$[\![e]\!](E) = E' \text{ si } \exists N, e', b, S.\ N \vdash e \xrightarrow[S]{E', b} e' \text{ avec } S^m = E$$

Le plus petit environnement de signaux qui définit la sémantique de e est tel que S^m est égal au plus petit point fixe de $[\![e]\!]$. On l'obtient en itérant cette fonction croissante à partir de l'événement vide. Le fait que l'on atteigne bien un point fixe est garanti par le lemme suivant :

Lemme 4.4. *Soit A un ensemble ordonné de minimum $\bot$. Si $f : A \to A$ est une fonction croissante, et si la suite $x_n = f^n(\bot)$ est constante égale à y à partir d'un certain rang, alors y est le plus petit point fixe de f. On note $y = \texttt{lfp}(f)$.*

Démonstration. Il est immédiat que y est un point fixe de f. En outre, comme la fonction est croissante, on peut montrer par récurrence que c'est son plus petit point fixe. Supposons que y' est un autre point fixe. Alors on a :

- $x_0 = \bot \leq y'$
- Si $x_i \leq y'$, alors $x_{i+1} = f(x_i) \leq f(y') = y'$ par croissance de la fonction

□

Cas de REACTIVEML avec domaines réactifs Un point important à noter est que ce calcul est possible car les règles de dérivation ne dépendent pas de l'absence d'un signal. Cela garantit que la fonction $[\![e]\!]$ est croissante, c'est-à-dire que plus l'on suppose de signaux présents, plus on aura de signaux effectivement émis. Ce n'est plus le cas lorsque l'on considère les domaines réactifs, comme dans l'exemple suivant :

```
let process present_fast =
  signal slow in
  domain ck do
    signal fast in
    present fast then () else emit slow
    ||
    emit fast
  done
```

Si l'on calcule le point fixe de façon globale, alors on va d'abord considérer un environnement de signaux dans lequel tous les signaux sont absents. Si on considère le corps du domaine réactif, alors le signal `fast` est absent, donc on doit exécuter la branche **`else`** à l'instant suivant de `ck`. `slow` est donc considéré comme présent pendant le premier instant de l'horloge globale. Si on avait supposé `fast` présent au premier instant de `ck`, alors `slow` serait absent. La fonction n'est donc plus croissante : en ajoutant plus de signaux présents, on obtient moins de signaux émis.

Pour corriger ce problème, on doit faire un point fixe localement pour chaque instant d'un domaine réactif. L'idée est que ce point fixe va donner le statut et la valeur des signaux pendant le premier instant de l'horloge locale. On suppose au cours de ce point fixe que le statut des signaux lents reste inchangé. Ceci est possible grâce à la restriction sur la dépendance instantanée sur un signal lent, qui assure que le statut des signaux lents n'influence pas le point fixe local. Une fois que l'on a calculé ce point fixe local, on doit choisir s'il faut ou non exécuter un autre instant local. Il suffit de regarder le statut de retour du corps du domaine et son compteur pour voir si l'on peut appliquer la règle COUNTEREOI ou PARENTEOI. Si c'est le cas, alors on peut continuer le point fixe de l'horloge parente. Sinon, il faut ajouter un environnement local de signaux pour l'instant suivant de l'horloge du domaine puis calculer le point fixe pour cet instant, et ainsi de suite. Nous ne rentrons pas dans les détails du calcul du point fixe qui devient complexe avec l'ajout des domaines réactifs.

CHAPITRE 5

Sémantique opérationnelle

Nous allons présenter dans ce chapitre une deuxième sémantique formelle de REACTIVEML étendu avec les domaines réactifs. Elle permet de donner une vision plus opérationnelle de l'exécution des programmes, avec notamment la construction de l'environnement de signaux et le choix du nombre d'instants que doit exécuter chaque domaine réactif. Nous montrerons dans un second temps l'équivalence des deux sémantiques. Enfin, nous comparerons notre approche basée sur les domaines réactifs avec les travaux similaires dans d'autres langages synchrones.

5.1 Sémantique opérationnelle

La sémantique comportementale que nous avons présentée au chapitre précédent ne donne pas de moyen direct pour exécuter les programmes. Nous avons vu dans la partie 4.5 qu'il faut calculer un point fixe sur tout le programme pour connaître l'environnement de signaux dans lequel effectuer la réaction. Nous allons maintenant présenter une sémantique opérationnelle qui montre comment exécuter les programmes de façon directe, en construisant cet environnement au cours de l'exécution. Elle servira de base à l'implémentation que nous décrirons dans le chapitre 9. Cette sémantique reprend la structure de celle de REACTIVEML [MP08b], qui est elle-même une extension de la sémantique à petits-pas de ML. Le principe est de distinguer deux phases dans l'exécution d'un instant et de définir une réduction par phase :

- La réduction d'instant définit le comportement du programme pendant l'instant.
- La réduction de fin d'instant prépare l'exécution de l'instant suivant.

L'exécution d'un instant du programme est une succession de pas de la réduction d'instant, suivi par un pas de la réduction de fin d'instant.

Attente automatique des domaines réactifs Avant de pouvoir donner la sémantique proprement dite, nous allons définir le prédicat d'attente de l'instant suivant, qui est la base de l'attente automatique des domaines réactifs. Il permet de savoir si le corps d'un domaine réactif est bloqué en attente d'une horloge plus lente ou s'il doit encore effectuer un pas sur l'horloge locale. Il est défini par :

$$s, C, \mathcal{S} \vdash_{\text{next}} e$$

qui signifie que dans la pile s et l'environnement de signaux $\mathcal{S}$, si l'on est à la fin de l'instant des horloges contenues dans l'ensemble C, alors l'expression e doit exécuter un autre instant de l'horloge locale.

Le prédicat est défini sur la figure 5.1 :

- Il y a trois cas principaux où une expression attend l'instant suivant : si on teste la présence d'un signal absent (NEXTPRESENT), si on vient de préempter le corps d'un `do/until` suite à la présence du signal $n^{ck'}$ (NEXTUNTIL) ou si on attend l'instant suivant avec `pause` ck' (NEXTPAUSE). Dans les deux derniers cas, il faut vérifier que l'horloge ck' dont on attend l'instant suivant appartient bien à l'ensemble C des horloges en fin d'instant.

$$(\textsc{NextPresent})\ \frac{n \notin \mathcal{S}(s)}{s, C, \mathcal{S} \vdash_{\text{next}} \texttt{present}\ n^{ck}\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$(\textsc{NextUntil})\ \frac{n \in \mathcal{S}(s_{\restriction ck'}) \qquad ck' \in C}{s, C, \mathcal{S} \vdash_{\text{next}} \texttt{do}\ e_1\ \texttt{until}\ n^{ck'}(x) \to e_2}$$

$$(\textsc{NextPause})\ \frac{ck' \in C}{s, C, \mathcal{S} \vdash_{\text{next}} \texttt{pause}\ ck'} \qquad (\textsc{NextIn})\ \frac{s :: (ck', i), C \cup \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1}{s, C, \mathcal{S} \vdash_{\text{next}} e_1\ \texttt{in}\ (ck', i/j)}$$

$$\frac{s, C, \mathcal{S} \vdash_{\text{next}} e_1}{s, C, \mathcal{S} \vdash_{\text{next}} \texttt{do}\ e_1\ \texttt{until}\ n^{ck'}(x) \to e_2} \qquad \frac{n \in \mathcal{S}(s) \qquad s, C, \mathcal{S} \vdash_{\text{next}} e_1}{s, C, \mathcal{S} \vdash_{\text{next}} \texttt{do}\ e_1\ \texttt{when}\ n^{ck}}$$

$$\frac{s, C, \mathcal{S} \vdash_{\text{next}} e_1}{s, C, \mathcal{S} \vdash_{\text{next}} \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3} \qquad \frac{s, C, \mathcal{S} \vdash_{\text{next}} e_2}{s, C, \mathcal{S} \vdash_{\text{next}} \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3}$$

FIGURE 5.1 – Prédicat d'attente de l'instant suivant

– On voit ici la différence entre `pause` ck' et `qpause` ck'. Dans le premier cas, on considère que l'expression attend l'instant suivant, alors que dans le second cas elle est considérée comme bloquée, puisqu'aucune règle ne mentionne `qpause` ck'. Cela reflète bien l'intuition de cette construction : elle attend le prochain instant d'une horloge, mais doit être considérée comme bloquée par le domaine réactif où elle s'exécute.
– Dans le cas d'un domaine réactif, on vérifie si le corps e_1 doit faire un autre instant, en ajoutant l'horloge locale ck' du domaine sur la pile et dans la liste des horloges en fin d'instant (NEXTIN).
– Les dernières règles sont structurelles. La construction `do`/`until` attend l'instant suivant si son corps l'attend. Pour une suspension `do`/`when`, on ne regarde le corps que si le signal n^{ck} est présent. S'il est absent, alors l'expression est bloquée. Dans le cas du `let`/`and`, il suffit qu'une des deux branches attende l'instant suivant.

Réduction d'instant Comme pour la sémantique comportementale, on caractérise la réduction d'instant par la pile s dans laquelle on exécute l'expression. Les notations utilisées dans cette partie sont les mêmes que dans le chapitre précédent (voir partie 4.2). La réduction d'instant est ainsi définie par :

$$e/\mathcal{S} \xrightarrow{s} e'/\mathcal{S}'$$

qui signifie que dans la pile s, l'expression e se réduit en e' et transforme l'environnement de signaux $\mathcal{S}$ en $\mathcal{S}'$. On note $\xrightarrow{s}{}^{*}$ la fermeture transitive de la réduction d'instant et $e/\mathcal{S} \not\xrightarrow{s}$ lorsqu'une expression ne peut pas se réduire selon la réduction d'instant.

Les règles définissant cette relation sont visibles sur la figure 5.2. Contrairement à la sémantique de REACTIVEML, on ne distingue plus la réduction de tête (notée $\to_\epsilon$ dans [MP08b]) de la réduction dans un contexte pour alléger les notations. Il faut noter également que, bien qu'on la présente en premier pour des raisons pédagogiques, la réduction d'instant utilise la réduction de fin d'instant.

La figure 5.2a donne les règles correspondant à la réduction de tête :
– L'opérateur `run` permet d'évaluer le corps d'une définition de processus.
– On peut réduire le corps d'un `let`/`and`/`in` lorsque les deux branches ont terminé, c'est-à-dire lorsqu'elles se réduisent en deux valeurs v_1 et v_2.

$$\lambda x.e_1\ v/\mathcal{S} \xrightarrow{s} e_1[x \leftarrow v]/\mathcal{S} \qquad \texttt{rec}\ x = e_1/\mathcal{S} \xrightarrow{s} e_1[x \leftarrow \texttt{rec}\ x = e_1]/\mathcal{S}$$

$$\texttt{run}\ (\texttt{process}\ e_1)/\mathcal{S} \xrightarrow{s} e_1/\mathcal{S} \qquad \texttt{let}\ x_1 = v_1\ \texttt{and}\ x_2 = v_2\ \texttt{in}\ e_3/\mathcal{S} \xrightarrow{s} e_3[x_1 \leftarrow v_1; x_2 \leftarrow v_2]/\mathcal{S}$$

$$\frac{n \notin \mathit{Sig}(\mathcal{S}) \qquad \mathcal{S}' = \mathcal{S} \sqcup \{s_{\restriction ck'} \mapsto \{n \mapsto (v_d, v_g, v_d, \emptyset, b, ck'')\}\}}{\texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : ck', \texttt{d} : v_d, \texttt{g} : v_g, \texttt{rck} : ck'')\ \texttt{in}\ e_1/\mathcal{S} \xrightarrow{s} e_1[x \leftarrow n^{ck'}]/\mathcal{S}'}$$

$$(\textsc{Emit})\frac{\mathcal{S}' = \mathcal{S} + \{s_{\restriction ck'} \mapsto \{n \mapsto \{v\}\}\}}{\texttt{emit}\ n^{ck'}\ v/\mathcal{S} \xrightarrow{s} ()/\mathcal{S}'} \qquad \frac{v = \mathcal{S}^l(s_{\restriction ck'})(n)}{\texttt{last}\ n^{ck'}/\mathcal{S} \xrightarrow{s} v/\mathcal{S}}$$

$$\frac{n \in \mathcal{S}(s)}{\texttt{present}\ n^{ck}\ \texttt{then}\ e_1\ \texttt{else}\ e_2/\mathcal{S} \xrightarrow{s} e_1/\mathcal{S}} \qquad \frac{}{\texttt{do}\ v\ \texttt{until}\ n^{ck'}(x) \to e_2/\mathcal{S} \xrightarrow{s} v/\mathcal{S}}$$

$$\frac{n \in \mathcal{S}(s)}{\texttt{do}\ v\ \texttt{when}\ n^{ck}/\mathcal{S} \xrightarrow{s} v/\mathcal{S}} \qquad \texttt{local\_ck}/\mathcal{S} \xrightarrow{s} \texttt{top}(s)/\mathcal{S}$$

$$(\textsc{Inst})\frac{j > 0 \qquad ck' \notin \mathit{Clocks}(\mathcal{S}) \qquad \mathcal{S}' = \mathcal{S} \sqcup \{s :: (ck', 0) \mapsto []\}}{\texttt{domain}\ x\ \texttt{by}\ j\ \texttt{do}\ e_1/\mathcal{S} \xrightarrow{s} (e_1[x \leftarrow ck'])\ \texttt{in}\ (ck', 0/j)/\mathcal{S}'}$$

$$(\textsc{Term})\frac{i < j}{v\ \texttt{in}\ (ck', i/j)/\mathcal{S} \xrightarrow{s} v/\mathcal{S}}$$

(a) Réductions de tête

$$(\textsc{Context})\frac{e_1/\mathcal{S} \xrightarrow{s} e_1'/\mathcal{S}'}{\Gamma(e_1)/\mathcal{S} \xrightarrow{s} \Gamma(e_1')/\mathcal{S}'} \qquad (\textsc{ContextWhen})\frac{n \in \mathcal{S}(s) \qquad e_1/\mathcal{S} \xrightarrow{s} e_1'/\mathcal{S}'}{\texttt{do}\ e_1\ \texttt{when}\ n^{ck}/\mathcal{S} \xrightarrow{s} \texttt{do}\ e_1'\ \texttt{when}\ n^{ck}/\mathcal{S}'}$$

$$(\textsc{LocalStep})\frac{i < j \qquad s' = s :: (ck', i) \qquad e_1/\mathcal{S} \xrightarrow{s'} e_1'/\mathcal{S}'}{e_1\ \texttt{in}\ (ck', i/j)/\mathcal{S} \xrightarrow{s} e_1'\ \texttt{in}\ (ck', i/j)/\mathcal{S}'}$$

$$(\textsc{LocalEoi})\frac{\begin{array}{c} i < j - 1 \qquad s' = s :: (ck', i) \qquad e_1/\mathcal{S} \not\xrightarrow{s'} \qquad s', \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1 \\ s', \{ck'\} \vdash e_1/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'/\mathcal{S}' \qquad \mathcal{S}'' = \mathcal{S}' \sqcup \{\texttt{snext}(s', \{ck'\}) \mapsto \texttt{next}_{\{ck'\}}(\mathcal{S}(s'))\} \end{array}}{e_1\ \texttt{in}\ (ck', i/j)/\mathcal{S} \xrightarrow{s} e_1'\ \texttt{in}\ (ck', i+1/j)/\mathcal{S}''}$$

(b) Réductions en contexte

FIGURE 5.2 – Réduction d'instant

– On s'assure que le nom n utilisé lors de la création d'un signal est frais en vérifiant qu'il n'apparait dans aucun des environnements locaux de signaux. Le nouveau signal a une valeur par défaut v_d, une fonction de combinaison v_g, sa dernière valeur est initialisée à v_d, le muti-ensemble des valeurs émises est vide, b indique s'il s'agit d'un signal à mémoire et ck'' est son horloge de réinitialisation. Contrairement à la sémantique comportementale, on ne devine plus l'ensemble des valeurs émises sur le signal. On commence par un multi-ensemble vide et on accumule les valeurs au fur et à mesure de la réduction, comme on le peut voir dans la règle EMIT, qui ajoute la valeur v dans le multi-ensemble des valeurs émises sur $n^{ck'}$.
– La lecture des signaux dans l'environnement se fait de la même façon que pour la sémantique comportementale, comme on peut le voir pour la règle du `last`. Il faut en particulier noter que l'opération $s_{\restriction ck'}$ de restriction d'une pile n'est définie que pour les horloges plus lentes que l'horloge locale $\texttt{top}(s)$. L'accès à un signal sur une horloge autre n'a donc pas non plus de sémantique opérationnelle.
– On ne réduit l'expression `present` que si le signal n^{ck} est présent. On exécute alors immédiatement l'expression e_1.
– Si le corps d'une préemption ou d'une suspension est une valeur v, alors l'expression se réécrit instantanément en la même valeur.
– L'expression `local_ck` se réduit instantanément en l'horloge locale $\texttt{top}(s)$ que l'on trouve au sommet de la pile courante.
– L'instanciation d'un domaine réactif (INST) se fait en créant un nom frais d'horloge ck' que l'on substitue à x dans le corps e_1 du domaine. On ajoute aussi à l'environnement de signaux un nouvel environnement local de signaux vide associé au premier instant de l'horloge locale, qui est désigné par la pile $s :: (ck', 0)$.
– Lorsque le corps e_1 du domaine se réduit en une valeur v, le domaine termine instantanément et se réduit en la même valeur (TERM).

La figure 5.2b montre les règles concernant les réductions dans un contexte :

– La première règle montre que si une expression se réduit, alors elle peut également se réduire dans n'importe quel contexte d'évaluation Γ (défini page 55). On remarque en particulier que le corps de la construction `do`/`when` n'est pas un contexte d'évaluation, puisqu'on ne peut exécuter son corps que si le signal n^{ck} est présent. On ajoute donc une règle pour ce cas précis.
– Une réduction d'un domaine réactif e_1 `in` $(ck', i/j)$ est donnée par une réduction de son corps e_1 en ajoutant l'horloge locale ck' et l'instant courant i sur la pile s (règle LOCALSTEP). Cela revient à exécuter le corps du domaine dans l'horloge locale.
– Lorsque l'on ne peut plus faire de réduction d'instant sur l'horloge locale, on vérifie si le corps demande un autre instant local à l'aide du prédicat $\vdash_{\text{next}}$. Si c'est le cas et si le compteur d'instants n'a pas atteint sa valeur maximale $j-1$, alors on effectue une réduction de fin d'instant locale (règle LOCALEOI). On prépare ensuite l'exécution de l'instant suivant en ajoutant un nouvel environnement local de signaux pour l'instant suivant de l'horloge locale, qui est désigné par la pile $\texttt{snext}(s', ck') = s :: (ck', i+1)$. On calcule le nouvel environnement local de signaux en prenant le successeur de l'environnement local de signaux correspondant à l'instant courant de l'horloge locale avec l'opération $\texttt{next}_C(S)$, qui est définie page 59.

Réduction de fin d'instant La réduction de fin d'instant effectue le passage à l'instant suivant. En effet, dans le modèle synchrone réactif de REACTIVEML, il faut attendre la fin de l'instant pour décider de l'absence d'un signal et connaître la valeur des signaux présents. Les signaux non émis sont alors considérés absents et la valeur des signaux est fixée, puisqu'il ne peut plus y avoir de nouvelle émission. On exécute donc la réduction de fin d'instant une fois que l'on ne peut plus effectuer de réduction d'instant. C'est pourquoi la réduction n'est donnée que pour certaines expressions, que l'on appelle *expressions de fin d'instant*. Les autres expressions sont réécrites par la réduction d'instant.

$$\frac{s, C \vdash e_1/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'/\mathcal{S}' \qquad s, C \vdash e_2/\mathcal{S}' \longrightarrow_{\text{eoi}} e_2'/\mathcal{S}''}{s, C \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3/\mathcal{S} \longrightarrow_{\text{eoi}} \texttt{let}\ x_1 = e_1'\ \texttt{and}\ x_2 = e_2'\ \texttt{in}\ e_3/\mathcal{S}''}$$

(EOIVALUE) $s, C \vdash v/\mathcal{S} \hookrightarrow_{\text{eoi}}$ $\qquad$ $\dfrac{n \notin \mathcal{S}(s)}{s, C \vdash \texttt{present}\ n^{ck}\ \texttt{then}\ e_1\ \texttt{else}\ e_2/\mathcal{S} \longrightarrow_{\text{eoi}} e_2/\mathcal{S}}$

$$\text{(EOIUNTILPRES)}\ \frac{n \in \mathcal{S}(s_{\restriction ck'}) \wedge ck' \in C \qquad v = \mathcal{S}^v(s_{\restriction ck'})(n)}{s, C \vdash \texttt{do}\ e_1\ \texttt{until}\ n^{ck'}(x) \to e_2/\mathcal{S} \longrightarrow_{\text{eoi}} e_2[x \leftarrow v]/\mathcal{S}'}$$

$$\text{(EOIUNTILABS)}\ \frac{not\left(n \in \mathcal{S}(s_{\restriction ck'}) \wedge ck' \in C\right) \qquad s, C \vdash e_1/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'/\mathcal{S}'}{s, C \vdash \texttt{do}\ e_1\ \texttt{until}\ n^{ck'}(x) \to e_2/\mathcal{S} \longrightarrow_{\text{eoi}} \texttt{do}\ e_1'\ \texttt{until}\ n^{ck'}(x) \to e_2/\mathcal{S}'}$$

$$\text{(EOIWHENPRES)}\ \frac{n \in \mathcal{S}(s) \qquad s, C \vdash e_1/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'/\mathcal{S}'}{s, C \vdash \texttt{do}\ e_1\ \texttt{when}\ n^{ck}/\mathcal{S} \longrightarrow_{\text{eoi}} \texttt{do}\ e_1'\ \texttt{when}\ n^{ck}/\mathcal{S}'}$$

$$\text{(EOIWHENABS)}\ \frac{n \notin \mathcal{S}(s)}{s, C \vdash \texttt{do}\ e_1\ \texttt{when}\ n^{ck}/\mathcal{S} \hookrightarrow_{\text{eoi}}}$$

(a) Réduction de fin d'instant

$$\text{(PAUSEEOI)}\ \frac{ck' \in \mathit{Clocks}(s) \qquad ck' \in C}{s, C \vdash \texttt{pause}\ ck'/\mathcal{S} \longrightarrow_{\text{eoi}} ()/\mathcal{S}} \qquad \text{(PAUSENOTEOI)}\ \frac{ck' \in \mathit{Clocks}(s) \qquad ck' \notin C}{s, C \vdash \texttt{pause}\ ck'/\mathcal{S} \hookrightarrow_{\text{eoi}}}$$

$$\text{(QPAUSEEOI)}\ \frac{ck' \in \mathit{Clocks}(s) \qquad ck' \in C}{s, C \vdash \texttt{qpause}\ ck'/\mathcal{S} \longrightarrow_{\text{eoi}} ()/\mathcal{S}} \qquad \text{(QPAUSENOTEOI)}\ \frac{ck' \in \mathit{Clocks}(s) \qquad ck' \notin C}{s, C \vdash \texttt{qpause}\ ck'/\mathcal{S} \hookrightarrow_{\text{eoi}}}$$

$$\text{(PARENTEOI)}\ \frac{\begin{array}{c} s' = s :: (ck', i) \qquad C' = C \cup \{ck'\} \qquad e_1/\mathcal{S} \overset{s'}{\not\longrightarrow} \qquad not(i < j - 1 \wedge s', \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1) \\ s', C' \vdash e_1/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'/\mathcal{S}' \qquad \mathcal{S}'' = \mathcal{S}' \sqcup \{\texttt{snext}(s', C') \mapsto \texttt{next}_{C'}(\mathcal{S}(s'))\} \end{array}}{s, C \vdash e_1\ \texttt{in}\ (ck', i/j)/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'\ \texttt{in}\ (ck', 0/j)/\mathcal{S}''}$$

(b) Réduction de fin d'instant des domaines réactifs

FIGURE 5.3 – Réduction de fin d'instant

La réduction de fin d'instant est définie par :

$$s, C \vdash e/\mathcal{S} \longrightarrow_{\text{eoi}} e'/\mathcal{S}'$$

qui signifie que dans la pile s et pendant la fin de l'instant des horloges contenues dans C, l'expression e se réduit en e' et transforme l'environnement $\mathcal{S}$ en $\mathcal{S}'$. On définit également :

$$s, C \vdash e/\mathcal{S} \hookrightarrow_{\text{eoi}} \triangleq\ s, C \vdash e/\mathcal{S} \longrightarrow_{\text{eoi}} e/\mathcal{S}$$

qui signifie que l'expression e est inchangée par la réduction de fin d'instant. Il faut bien garder en tête que, même si la réduction de fin d'instant renvoie un nouvel environnement de signaux $\mathcal{S}'$, aucune valeur ne peut être émise pendant la fin de l'instant. La seule modification de l'environnement de signaux faite pendant la fin de l'instant est d'ajouter des nouveaux environnements locaux de signaux pour les prochains instants des horloges rapides.

Les règles définissant la réduction de fin d'instant des expressions de base du langage sont données sur la figure 5.3a. Elles reprennent les grandes lignes de celles de REACTIVEML :

- On applique récursivement la réduction dans les deux branches de la composition parallèle. Si une seule des branches du parallèle a terminé son exécution, c'est-à-dire qu'elle s'est réduite en une valeur, alors elle reste inchangée par la réduction de fin d'instant jusqu'à ce que l'autre branche termine (EOIVALUE).
- On réécrit l'expression `present` si le signal n^{ck} est absent. On regarde sa présence dans l'environnement local de signaux $\mathcal{S}(s)$. On sait en effet qu'il s'agit d'un signal sur l'horloge locale $\texttt{top}(s)$ puisque le test de présence constitue une dépendance instantanée.
- On peut noter également que, dans le cas de la préemption, on ne réécrit l'expression que si le signal $n^{ck'}$ est présent et si l'on est dans la fin de l'instant de son horloge ck' (EOIUNTILPRES). Dans le cas contraire, on réécrit le corps e_1 tout en conservant la préemption (EOIUNTILABS).
- On n'effectue la fin de l'instant du corps e_1 du `do`/`when` que si le signal de suspension est présent (EOIWHENPRES). Sinon, l'expression reste inchangée (EOIWHENABS).

La figure 5.3b montre les règles en rapport avec les domaines réactifs :

- Les expressions `pause` ck' et `qpause` ck' ont le même comportement et ne diffèrent que pour le prédicat $\vdash_{\text{next}}$. Elles se réécrivent quand on est à la fin de l'instant de l'horloge ck', c'est-à-dire lorsque ck' appartient à l'ensemble C (PAUSEEOI et QPAUSEEOI). Dans le cas contraire, l'expression reste inchangée mais on vérifie tout de même que ck' fait bien partie des horloges accessibles, c'est-à-dire des horloges apparaissant dans la pile s (PAUSENOTEOI et QPAUSENOTEOI).
- Dans le cas des domaines réactifs, on a une seule règle (PARENTEOI). Elle s'applique lorsque le domaine ne peut plus se réduire sur l'horloge locale et attend le prochain instant d'une horloge plus lente. On effectue alors une réduction de fin d'instant du corps e_1 du domaine en ajoutant l'horloge locale ck' sur la pile et dans l'ensemble C des horloges en fin d'instant. On remet ensuite à zéro le compteur d'instants du domaine et on prépare l'exécution de l'instant suivant en ajoutant un nouvel environnement local de signaux associé à cet instant suivant, que l'on calcule avec l'opération $\texttt{snext}(s', C')$ définie page 57.

Réaction d'un programme La réaction du programme, que l'on note $e_i/\mathcal{S}_i \overset{I_i}{\underset{O_i}{\Rightarrow}} e''_i/\mathcal{S}''_i$, consiste à appliquer autant de fois que possible la réduction d'instant dans le domaine réactif d'horloge $\top_{ck}$. On applique ensuite une fois la réduction de fin d'instant :

$$\frac{\begin{array}{c} s_i = (\top_{ck}, i) \qquad e_i/\mathcal{S}_i \xrightarrow{s_i}{}^{*} e'_i/\mathcal{S}'_i \qquad e'_i/\mathcal{S}'_i \not\xrightarrow{s_i} \\ s_i, \{\top_{ck}\} \vdash e'_i/\mathcal{S}'_i \longrightarrow_{\text{eoi}} e''_i/\mathcal{S}''_i \qquad I_i = \mathcal{S}_i^v((\top_{ck}, i))\ (1) \qquad O_i \subseteq \mathcal{S}'^v_i((\top_{ck}, i))\ (2) \end{array}}{e_i/\mathcal{S}_i \overset{I_i}{\underset{O_i}{\Rightarrow}} e''_i/\mathcal{S}''_i}$$

Les conditions supplémentaires signifient que :

(1) Les seuls signaux présents dans l'environnement de signaux au début de l'instant sont les entrées I_i.
(2) Les sorties O_i sont lues dans l'environnement de signaux à la fin de l'exécution de l'instant.

On note $e_i/\mathcal{S}_i \Rightarrow e'_i/\mathcal{S}'_i$ lorsque $I_i = \{\}$ et $O_i = \{\}$.

Remarque 3. [MP13b] montre une approche un peu différente de la sémantique opérationnelle du langage. Alors que l'on a cherché ici à se rapprocher de la sémantique comportementale du chapitre 5, la sémantique de l'article utilise un environnement de signaux plus simple, qui ne stocke que les environnements locaux de signaux correspondant à l'instant courant. La gestion des horloges accessibles ne se fait plus par la pile mais par un environnement d'horloges $\mathcal{H}$, qui associe à chaque horloge son horloge parente et les valeurs courantes et maximales du compteur d'instants. Le compteur d'instants indique le nombre d'instants restants au lieu du nombre d'instants déjà faits. On peut remarquer également que dans cet article, on teste si le domaine réactif doit faire un autre instant local en regardant si la réduction de fin d'instant modifie ou non le corps du domaine. Autrement dit, $s, C, \mathcal{S} \vdash_{\text{next}} e$ est défini par $s, C \vdash e/\mathcal{S} \longrightarrow_{\text{eoi}} e'/\mathcal{S}'$ avec $e \neq e'$. Les deux définitions sont équivalentes pour un langage sans la construction `qpause`. Il s'agit en effet de la seule construction qui se réécrit en une expression différente lors de la fin de l'instant mais sans demander à s'exécuter à l'instant suivant.

Remarque 4. La sémantique opérationnelle que nous venons de présenter peut être étendue pour prendre en compte les références. On peut suivre la démarche suivie pour REACTIVEML (partie 4.3.2 de [Man06]), qui consiste à ajouter dans la réduction d'instant un état mémoire M, qui associe une valeur à une location mémoire. Les règles pour la création, la lecture et la modification d'une référence sont les mêmes que dans le cas de REACTIVEML.

5.2 Équivalence des sémantiques

Nous allons maintenant montrer l'équivalence entre la sémantique comportementale et la sémantique opérationnelle. Nous verrons dans le chapitre 6 que la preuve de sûreté du calcul d'horloges se base sur la sémantique opérationnelle, alors que l'on utilise la sémantique comportementale pour montrer le déterminisme de la sémantique (Propriété 4.1) et la sûreté de l'analyse de réactivité du chapitre 7.

Nous reprenons l'approche de la preuve d'équivalence des sémantiques de REACTIVEML [MP08b]. On commence par montrer que si une expression e peut se réduire en une expression e' dans la sémantique opérationnelle, alors elle peut réagir dans le même environnement de signaux dans la sémantique comportementale. On doit pour cela montrer plusieurs lemmes. Le premier fait le lien entre le prédicat $\vdash_{\text{next}}$ et le statut d'une expression dans la sémantique comportementale.

Lemme 5.1. *Si une expression de fin d'instant e est telle que $s, C, \mathcal{S} \not\vdash_{next} e$, alors son statut de retour dans la dérivation de la sémantique comportementale n'appartient pas à l'ensemble C des horloges en fin d'instant. Formellement :*

$$\left(N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e' \ \wedge \ s, C \vdash e/\mathcal{S} \longrightarrow_{eoi} e'/\mathcal{S}'\right) \Rightarrow (s, C, \mathcal{S} \not\vdash_{next} e \Rightarrow k \notin C)$$

Démonstration. Nous allons en fait montrer la contraposée de l'implication de droite, c'est-à-dire que si le statut de retour k appartient à C, alors l'expression attend l'instant suivant :

$$\left(N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e' \ \wedge \ s, C \vdash e/\mathcal{S} \longrightarrow_{\text{eoi}} e'/\mathcal{S}'\right) \Rightarrow (k \in C \Rightarrow s, C, \mathcal{S} \vdash_{\text{next}} e)$$

On raisonne par induction sur la dérivation de la sémantique comportementale de e :

Cas `let` $x_1 = e_1$ `and` $x_2 = e_2$ `in` e_3. Puisque $k = \min_s(k_1, k_2) \in C$, on sait que k_1 ou k_2 est dans C. Supposons que ce soit k_1. Alors on peut appliquer l'hypothèse d'induction sur e_1, qui permet de conclure que $s, C, \mathcal{S} \vdash_{\text{next}} e_1$. On en déduit le résultat attendu en utilisant le premier cas du `let`/`and`/`in` dans la figure 5.1.

Cas `present` e_s `then` e_1 `else` e_2**.** Comme e est une expression de fin d'instant, on sait que le signal est absent, c'est-à-dire que $n \notin \mathcal{S}(s)$. On peut donc conclure en appliquant la règle NEXTPRESENT.

Cas `do` e_1 `until` $e_s(x) \to e_2$**.** On doit considérer plusieurs cas :
- Si $n \notin \mathcal{S}(s_{\restriction ck'})$, alors on est dans le cas de la règle UNTILABS. On peut donc appliquer l'hypothèse d'induction sur le corps e_1 et conclure simplement.
- Si $n \in \mathcal{S}(s_{\restriction ck'})$ (règle UNTILPRESEOI), alors on a $ck' \in C$ qui nous permet d'appliquer la règle NEXTUNTIL.
- Si $n \in \mathcal{S}(s_{\restriction ck'})$ (règle UNTILPRESNOTEOI), alors soit k est égal à l'horloge ck' du signal et on conclut comme dans le second cas, soit k est le statut ck'' du corps du domaine et on conclut comme dans le premier cas.

Cas e_1 `in` $(ck', i/j)$**.** On peut appliquer l'hypothèse d'induction sur le corps e_1 du domaine avec $C' = C \cup \{ck'\}$ et $s' = s :: (ck', i)$. En effet, le statut de retour de e_1 est soit égal à celui de e, soit égal à l'horloge locale ck'. Dans les deux cas, il appartient bien à C'. On obtient alors $s', C', \mathcal{S} \vdash_{\text{next}} e_1$, qui est bien l'hypothèse de la règle NEXTIN.

□

On montre maintenant que si une expression peut effectuer une réduction de fin d'instant, alors elle peut se réduire de la même façon dans la sémantique comportementale.

Lemme 5.2. *Si* $s, C \vdash e/\mathcal{S} \longrightarrow_{eoi} e'/\mathcal{S}'$ *avec* $C \subseteq Clocks(s)$ *alors il existe* N *et* k *tels que* $N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'$.

Démonstration. Par induction sur la dérivation de $s, C \vdash e/\mathcal{S} \longrightarrow_{\text{eoi}} e'/\mathcal{S}'$. Le cas délicat est celui du domaine réactif en cours d'exécution e_1 `in` $(ck', i/j)$ avec la règle PARENTEOI. On peut appliquer l'hypothèse d'induction sur e_1 qui se réécrit en e_1' en prenant l'ensemble d'horloges $C' = Clocks(s :: (ck', i))$ et la pile $s' = s :: (ck', i)$. La règle PARENTEOI donne la condition $not(i < j - 1 \wedge s', \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1)$. On a donc deux cas :
- Soit $i = j - 1$: On est dans le cas où le domaine a exécuté le nombre maximal d'instants locaux et attend le prochain instant de son horloge parente. On peut alors appliquer la règle COUNTEREOI.
- Soit $s', \{ck'\}, \mathcal{S} \not\vdash_{\text{next}} e_1$: On est cette fois dans le cas où le corps du domaine attend une horloge lente. Alors en appliquant le lemme 5.1, on sait que le statut de e_1 n'est pas égal à ck', donc on peut appliquer la règle PARENTEOI.

□

Le dernier lemme montre que l'on peut « composer » un pas de réduction d'instant avec la sémantique comportementale. Cela signifie que si e se réduit en e' par la réduction d'instant et que e' se réécrit en e'' dans la sémantique comportementale, alors e se réécrit également en e'' dans la sémantique comportementale.

Lemme 5.3. *Si* $e/\mathcal{S}_0 \xrightarrow{s} e'/\mathcal{S}_1$ *et* $N, s \vdash e' \xrightarrow[\mathcal{S}]{\mathcal{E}',k} e''$ *avec* $\mathcal{S}_1 \subseteq \mathcal{S}$*, alors* $N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e''$ *avec* $\mathcal{E} = \mathcal{E}' \sqcup (\mathcal{S}_1^m \setminus \mathcal{S}_0^m)$.

Démonstration. On fait la preuve par induction sur la dérivation de la sémantique comportementale. On s'intéresse particulièrement au cas des domaines réactifs :

Cas e_1 `in` $(ck', i/j)$ **avec la règle LOCALSTEP.** Il s'agit du cas où l'on exécute une réduction d'instant sur l'horloge locale pour réécrire e_1 en e_1'. On doit utiliser l'hypothèse d'induction sur l'horloge locale du domaine, puis reconstruire la dérivation du domaine. Plus précisément, on a alors $\mathcal{S}_0 = \mathcal{S}$, $e' = e_1'$ `in` $(ck', i/j)$ et $\mathcal{S}_1 = \mathcal{S}'$ avec $e_1/\mathcal{S} \xrightarrow{s'} e_1'/\mathcal{S}'$. On a également $N, s \vdash e_1'$ `in` $(ck', i/j) \xrightarrow[\mathcal{S}]{\mathcal{E}',k} e''$ par hypothèse. On peut remarquer que dans toutes les règles de la sémantique comportementale, il existe alors k et e_1'' tels que $N, s' \vdash e_1' \xrightarrow[\mathcal{S}]{\mathcal{E}',k} e_1''$.

On peut donc appliquer l'hypothèse d'induction sur e_1, qui nous permet de conclure que $N, s' \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}',k} e_1''$. On conclut alors en reconstruisant la dérivation de la sémantique comportementale, en remplaçant e_1' par e_1.

Cas e_1 `in` $(ck', i/j)$ avec la règle LOCALEOI. La réduction consiste maintenant à effectuer la fin de l'instant de l'horloge locale. On utilise le lemme 5.2 pour conclure. Plus précisément, on a maintenant $s', \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1$ et $s', \{ck'\} \vdash e_1/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'/\mathcal{S}'$. On utilise alors le lemme 5.2 pour en déduire que $N, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}',k} e_1'$. On peut alors utiliser la règle LOCALEOI pour reconstruire la dérivation complète.

□

On peut maintenant énoncer la première propriété, qui exprime le fait que si un programme se réécrit suivant la sémantique opérationnelle, alors il admet une dérivation dans la sémantique comportementale et se réécrit de la même façon.

Propriété 5.4. *Pour toute configuration $e/\mathcal{S}$, si $e/\mathcal{S} \Rightarrow e'/\mathcal{S}'$, alors il existe N, $\mathcal{E}$ et k tels que $N, (\top_{ck}, i) \vdash e \xrightarrow[\mathcal{S}']{\mathcal{E},k} e'$ avec $\mathcal{E} = \mathcal{S}' \setminus \mathcal{S}$.*

Démonstration. On procède par récurrence sur le nombre de réductions d'instant nécessaires :

- Le cas de base correspond au cas où l'on ne fait aucune réduction d'instant et une réduction de fin d'instant, c'est-à-dire que $(\top_{ck}, i), \{\top_{ck}\} \vdash e/\mathcal{S} \longrightarrow_{\text{eoi}} e'/\mathcal{S}'$. On conclut en utilisant le lemme 5.2.
- Supposons maintenant que l'on effectue $n+1$ pas de la réduction d'instant puis un pas de la réduction de fin d'instant. Autrement dit, on a :

$$e/\mathcal{S} \xrightarrow{(\top_{ck},i)} e_1/\mathcal{S}_1 \xrightarrow{(\top_{ck},i)} \dots \xrightarrow{(\top_{ck},i)} e_{n+1}/\mathcal{S}_{n+1} \quad \text{et} \quad (\top_{ck}, i), \{\top_{ck}\} \vdash e_{n+1}/\mathcal{S}_{n+1} \longrightarrow_{\text{eoi}} e'/\mathcal{S}'$$

On peut alors appliquer l'hypothèse de récurrence sur e_1 dont la réduction ne prend que n pas. On utilise ensuite le lemme 5.3 pour ajouter un pas de réduction d'instant supplémentaire et conclure.

□

On peut maintenant énoncer le théorème d'équivalence des deux sémantiques (nous ne considérons pas les entrées et sorties du programme pour simplifier l'énoncé du théorème) :

Théorème 5.5 (Équivalence des sémantiques comportementale et opérationnelle)**.** *Pour tout $\mathcal{S}$ et e tels que :*

- $N, (\top_{ck}, i) \vdash e \xrightarrow[\mathcal{S}_1]{\mathcal{E}_1,k_1} e_1$ *où $\mathcal{S}_1$ est le plus petit environnement tel que $\mathcal{S} \subseteq \mathcal{S}_1$*
- $e/\mathcal{S} \Rightarrow e_2/\mathcal{S}_2$
- *Toutes les fonctions de combinaison des signaux sont commutatives et associatives*

Alors $e_1 = e_2$ et $\mathcal{S}_1 = \mathcal{S}_2$.

Intuition de la démonstration. La propriété 5.4 étant la même que celle qui est énoncée dans [MP08b], la preuve de ce théorème est la même que pour REACTIVEML classique. On utilise pour cela la propriété 5.4 et les propriétés d'unicité (Propriété 4.2) et de déterminisme (Propriété 4.1) de la sémantique comportementale. □

Remarque 5. Compte tenu du très grand nombre de cas à considérer dans cette preuve d'équivalence, il serait intéressant d'utiliser un assistant de preuves pour s'assurer que l'on n'a pas oublié un cas intéressant. Cela permettrait en particulier de vérifier si les définitions de la partie 4.2 et les hypothèses des différents lemmes sont correctes. Nous n'avons pas mené ce travail au cours de cette thèse puisque la quantité de travail apparait comme démesurée par rapport à l'importance de ce théorème.

5.3 Travaux similaires

Le raffinement d'horloges en QUARTZ

La notion de domaines réactifs est proche de celle de *raffinement d'horloges* [GBS09, GBS13] dans le langage synchrone QUARTZ [Sch09]. L'idée de départ est la même : il s'agit d'introduire une notion d'instant local, que des processus peuvent utiliser pour se synchroniser et communiquer localement sur un rythme plus rapide. Notre travail présente toutefois des possibilités accrues de communication et de synchronisation entre des processus sur des horloges différentes et peut être appliqué dans un cadre plus général, notamment en présence de création dynamique, d'ordre supérieur et de signaux et processus de première classe.

Nous allons illustrer le raffinement d'horloges avec le processus suivant écrit dans le langage QUARTZ, avec le code REACTIVEML correspondant à droite :

```
module imm_dep(nat ?a, o)
{
  nat b;
  clock(C1) {
    nat x = b;
    l1: pause(C1);
    o = x + 3
  }
  ||
  if (a > 0) then b = 4
}
```

```
let process imm_dep a o =
  signal b in
  domain c1 do
    await immediate one b(x) in
    pause c1;
    emit o (x + 3)
  done
  ||
  if last a > 0 then emit b 4
```

On définit ici un processus `imm_dep` prenant en entrée un flot d'entiers `a` et retournant un flot d'entiers `o`, en définissant deux signaux locaux `b` et `x`. La construction **`clock`** permet de définir une horloge locale `C1`. On peut ensuite déclarer un signal `x` d'horloge `C1` en le définissant à l'intérieur du bloc, alors que les autres signaux sont sur l'horloge globale. On peut aussi attendre l'instant suivant de `C1` en appelant **`pause`**`(C1)`, auquel on associe une étiquette l_1. Le raffinement d'horloges autorise la dépendance instantanée sur un signal lent. Dans l'exemple, le corps du bloc dépend instantanément de la valeur du signal lent `b`. Le principe de l'exécution de ce programme est qu'au moment où on cherche à lire `b` à l'intérieur du bloc d'horloge `C1`, on ne connait pas encore sa valeur. Le bloc tout entier est donc bloqué. Ce n'est que lorsque l'on exécute la seconde branche du parallèle que l'on connait la valeur de `b` : si `a > 0`, alors `b` est présent et vaut `3`, sinon il est absent et vaut sa valeur précédente, c'est-à-dire `0` dans ce cas. On peut maintenant exécuter le premier instant de l'horloge `C1`, puis son second instant. On émet donc au cours du premier instant de l'horloge globale la valeur `7 = 4 + 3` sur la sortie `o` si `a > 0`, et `3 = 0 + 3` sinon.

Le raffinement d'horloges ne peut pas s'appliquer dans le cadre dynamique de REACTIVEML. En effet, puisque l'on peut créer dynamiquement des processus et puisque les signaux sont des objets de première classe qui peuvent échapper de leur portée lexicale, il est impossible de connaître les émetteurs potentiels d'un signal et de décider de son absence. Dans l'exemple ci-dessus, on n'est jamais sûr que le signal `b` ne sera pas émis plus tard dans l'instant et on ne peut donc pas terminer l'exécution du premier instant de `c1`. Nous avons donc choisi d'interdire la dépendance instantanée sur un signal lent, ce qui fait que le processus `imm_dep` n'a pas de sémantique et est rejeté par le calcul d'horloges que nous verrons dans le chapitre 6. Cette restriction permet également d'éviter les problèmes de causalité, comme par exemple avec le processus `immediate_dep_wrong` (page 36).

L'autre limite de l'approche de [GBS09] est qu'elle ne permet pas d'attendre l'émission d'un signal avec la construction `await` à l'intérieur de l'équivalent d'un domaine réactif. Cela pose un sérieux problème d'expressivité en restreignant fortement ce que l'on peut écrire dans le corps d'un domaine. La majorité des exemples que l'on a montré, comme la simulation des n-corps ou de réseau de capteurs, ne pourrait pas être écrits avec cette approche. Le problème est que

l'instruction `await` est encodée en QUARTZ par : [1]

$$\texttt{await}\ e_1(x)\ \texttt{in}\ e_2 \triangleq \texttt{do}\ (\texttt{loop}\ (\texttt{pause}\ \texttt{local\_ck}))\ \texttt{until}\ e_1(x) \rightarrow e_2$$

alors que nous avons choisi l'encodage suivant :

$$\texttt{await}\ e_1(x)\ \texttt{in}\ e_2 \triangleq \texttt{do}\ (\texttt{loop}\ (\texttt{qpause}\ \texttt{local\_ck}))\ \texttt{until}\ e_1(x) \rightarrow e_2$$

La différence avec notre encodage est que celui de QUARTZ utilise l'opérateur `pause` au lieu de `qpause`. La conséquence est qu'en QUARTZ, un domaine réactif dans lequel on attend un signal devient non coopératif, c'est-à-dire qu'il exécute un nombre infini d'instants locaux. Nous avons résolu ce problème en voyant un domaine réactif comme une entité propre qui décide d'attendre l'instant suivant de son horloge parente lorsque tous les processus qu'elle contient sont bloqués. Nous montrerons également dans le chapitre 7 une analyse de réactivité qui permet de détecter les domaines réactifs potentiellement non coopératifs.

Remarque 6. C'est bien l'attente automatique des domaines réactifs qui est importante, et pas la présence de l'opérateur `qpause`. On aurait aussi pu donner directement la sémantique de `await`, comme on l'a fait dans la partie 4.4 et dans [MPP13].

Enfin, nous verrons dans le chapitre 9 que l'on peut compiler les domaines réactifs de façon modulaire, ce qui n'est pas le cas de la méthode présentée dans [GBS10a]. On peut ainsi créer des domaines réactifs de façon dynamique et les imbriquer de façon arbitraire. Au contraire, la compilation du raffinement d'horloges ne s'applique que pour un arbre d'horloges statique connu à la compilation.

Les langages synchrones réactifs à la REACTIVEC

Micro-instants Le langage REACTIVEC [Bou91], le précurseur du modèle de concurrence de REACTIVEML, possède une notion de *micro-instants* qui est absente en REACTIVEML. L'idée est de pouvoir ordonner des effets de bords en subdivisant un instant en plusieurs micro-instants. Pour cela, on dispose de l'instruction `suspend`, qui permet d'attendre le micro-instant suivant, et de l'instruction `close` e qui exécute son corps e tant qu'il appelle `suspend`. On peut voir les micro-instants comme une version limitée de domaines réactifs, puisqu'on ne peut avoir que deux niveaux d'instants (instants et micro-instants) et que l'on ne peut pas communiquer dans les micro-instants en utilisant des signaux, car ceux-ci n'ont qu'une seule valeur par instant.

La possibilité de subdiviser un instant est l'un des exemples typiques d'utilisation des domaines réactifs que l'on a présenté dans le chapitre 3 (page 42). On pourrait donc chercher à encoder les micro-instants à l'aide de domaines réactifs de la manière suivante :

$$\texttt{close}\ e \triangleq \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e$$
$$\texttt{suspend} \triangleq \texttt{pause}\ \texttt{local\_ck}$$

Cette traduction naturelle n'est pas correcte du fait de l'impossibilité de dépendre instantanément d'un signal lent. En effet, l'expression `close` $(\texttt{await immediate}\ s)$ sera traduite en `domain` x `by` ∞ `do` $(\texttt{await immediate}\ s)$ qui n'a pas de sémantique car le corps du domaine dépend instantanément du signal lent s. Ce problème d'interaction entre les micro-instants et l'attente instantanée est d'ailleurs présent en REACTIVEC, où l'expression précédente boucle indéfiniment puisqu'une expression attendant un signal est considérée comme suspendue. La solution à ce problème a été apportée dans SUGARCUBES v3 [Sus01] en distinguant les suspensions et les signaux en attente. On peut alors donner un sens au programme suivant, qui reprend l'exemple `immediate_dep_wrong` (page 36) en utilisant des micro-instants :

1. On reprend ici la syntaxe de REACTIVEML pour simplifier la comparaison.

(a) Avant (b) Après

FIGURE 5.4 – Le raffinement temporel avec le sous-échantillonnage

```
let process immediate_dep_sugar =
  signal s in
  close (
    await immediate s; print_string "Ok"
    ||
    suspend; emit s
  )
```

Lors du premier micro-instant, la première branche du parallèle est en attente du signal `s` et la seconde est suspendue. On exécute ensuite le second micro-instant, au cours duquel `s` est émis, ce qui a pour conséquence l'affichage de `"Ok"`. Si on utilise l'encodage donné précédemment, alors ce programme est rejeté par le calcul d'horloges car il existe une dépendance instantanée sur le signal lent `s`. Mais si on utilise des micro-instants, on peut l'accepter car les micro-instants ne sont pas des vrais instants. Ils permettent d'avoir un contrôle sur l'ordonnancement des processus au cours de l'instant, mais sans changer la sémantique du programme.

Les machines réactives en SUGARCUBES Les SUGARCUBES [BS98] présentent une autre particularité proche des domaines réactifs, appelée *machine réactive*. Une machine réactive définit également une notion d'instant local, mais il faut lancer l'exécution de ses instants locaux manuellement. Pour cela, on appelle sa méthode `react`, qui exécute un instant de l'horloge locale, autant de fois que nécessaire. Dans le cas des domaines réactifs, ceci est fait de façon automatique.

En outre, les machines réactives ne peuvent communiquer entre elles que de façon limitée. On peut émettre un signal lent depuis une machine réactive, mais aussi émettre un signal rapide depuis l'extérieur de la machine. Il sera alors présent pour le prochain instant de la machine, c'est-à-dire la prochaine fois qu'on appelle sa méthode `react`. Par contre, on ne peut pas attendre l'émission d'un signal lent à l'intérieur d'une machine réactive ni attendre le prochain instant d'une horloge lente.

Suréchantillonnage dans les langages synchrones flot de données

La technique habituelle pour effectuer un raffinement temporel dans un langage synchrone flot de données est d'utiliser du *suréchantillonnage*. C'est par exemple la méthode choisie par [MC05] dans le langage LUSTRE, ou encore pour l'exemple 4 de [LGTLL03] en SIGNAL. L'idée du suréchantillonnage est de permettre à un processus d'effectuer plusieurs instants locaux rapides en ralentissant tous les autres, notamment avec l'opérateur **when** de filtrage. C'est ce que montre la figure 5.4.

Suréchantillonnage en LUSTRE Supposons que l'on définisse un nœud LUSTRE simple qui ajoute 2 à son entrée :

```
node add2(a:int) returns (o:int);
let
  o = a + 2;
tel
```

a	3	8	16	...
o	0	5	10	...

On cherche maintenant à raffiner ce nœud en prenant deux instants pour faire deux incrémentations :

```
node add2_ref(c:bool; a:int when c) returns (o:int when c);
var x:int;
let
  x = if c then (current(a) + 1) else (0 -> pre x + 1);
  o = (0 -> pre x) when c;
tel

node add2_ref_main(a:int) returns (o:int);
var c:bool;
let
  c = true -> not (pre c);
  o = current (add2_ref(c, a when c));
tel
```

Le nœud `add2_ref` prend maintenant une entrée `a` qui n'est présente que lorsque la condition `c` est vraie, c'est-à-dire un instant sur deux. Sa sortie `o` est elle aussi présente un instant sur deux.

c	*tt*	*ff*	*tt*	*ff*	*tt*	...
a	3	.	8	.	16	...
x	4	5	9	10	17	...
o	0	.	5	.	10	...

Remarque 7. En LUCID SYNCHRONE v1, il est possible d'écrire le programme précédent plus simplement, en intégrant le calcul de la condition `c` à l'intérieur du nœud `add2_ref`, puis en retournant cette condition en sortie du nœud. Cela permet d'exprimer le suréchantillonnage de façon plus locale et naturelle.

Suréchantillonnage en SIGNAL En LUSTRE, le suréchantillonnage est une transformation globale du programme. On peut l'exprimer de façon plus locale en SIGNAL, où le nœud `ADD2` s'écrit :

```
process ADD2 = (? integer a; ! integer o;)
  (| o := a + 2 |)
```

La version avec raffinement est la suivante :

```
process ADD2_REF = (? integer a ! integer o)
  (| o := (x$1 init 0) when c
   | x := (a + 1) default ((x+1)$1 init 0)
   | c := (not c)$1 init true
   | a ^= when c
   | c ^= x
   |)
where integer x; boolean c end
```

L'expression `e$1 init e_0` est équivalente à `e_0 -> pre e` en LUSTRE : elle vaut `e_0` au premier instant, puis la valeur précédente de `e` aux instants suivants. L'expression `e_1 default e_2` est égale à `e_1` lorsque `e_1` est présent, et sinon à `e_2`. Elle est donc présente si `e_1` ou `e_2` est présente, et est absente si les deux sont absentes. Les deux dernières lignes sont des contraintes d'horloges : on indique que l'horloge de `a`, que l'on note aussi `^a`, est égale à celle de `when c`, c'est-à-dire que `a` est présent lorsque `c` est présent et vrai. On exprime enfin le fait que `c` et `x` doivent être synchrones.

Cette version SIGNAL est équivalente à la version LUSTRE et on obtient exactement le même chronogramme. La différence principale est que cette version est plus naturelle, puisque l'on n'a pas besoin d'indiquer que l'entrée `a` et la sortie `o` ne sont pas présentes à tous les instants. C'est le compilateur SIGNAL qui se charge de la transformation de programme que l'on a faite à la main en LUSTRE. Il calcule en effet que l'horloge la plus rapide du programme est celle de `c` et de `x`, et que le reste du programme doit être exécuté seulement lorsque `c` est vrai.

Deux suréchantillonnages en parallèle La première limite du suréchantillonnage est que l'on peut réduire n instants locaux en un instant d'une horloge plus lente, mais on ne peut pas rendre le calcul instantané. On ne cache donc jamais complètement la présence d'instants locaux. C'est possible avec les domaines réactifs, puisque l'on peut en effet écrire les processus suivants :

```
let process add2 a o =
  loop await a(v) in emit o (v+2) end

let process add2_ref a o =
  domain ck do
    signal x in
    loop await a(v) in emit x (v+1) end
    ||
    loop await x(v) in emit o (v+1) end
  done
```

Grâce aux domaines réactifs, le processus `add2_ref` est équivalent à `add2`. La présence d'instants locaux est complètement invisible de l'extérieur, contrairement à LUSTRE et SIGNAL. Mais puisque REACTIVEML est un langage dynamique, la lecture des signaux prend un instant. On ne peut pas ici utiliser la construction `await immediate one` (qui permet de récupérer immédiatement une des valeurs émises sur un signal) car il s'agit d'une dépendance instantanée, que l'on interdit dans un domaine réactif. Nous verrons dans le chapitre 10 que l'on peut passer outre ces limitations dans le cadre plus statique d'ESTEREL. Le processus `levelorder_inst` (page 40) montre un autre cas où l'on peut utiliser un domaine réactif pour réduire plusieurs instants locaux en un processus instantané, ce que ne permet pas le suréchantillonnage.

L'autre limite du suréchantillonnage est qu'il s'agit d'une transformation de programmes non modulaire. Cela pose de sérieux problèmes si l'on souhaite mettre en parallèle deux processus qui font du suréchantillonnage. Si jamais ils font le même nombre d'instants locaux, par exemple si on lance en parallèle deux instances du nœud `add2_ref`, alors tout se passe bien. Mais on on ne peut pas écrire un programme dans lequel les deux processus font un nombre d'instants locaux différents. Cela ne pose aucun problème avec les domaines réactifs qui masquent complètement leurs instants locaux.

Supposons par exemple que l'on ait écrit un processus `add3_ref` sur le même modèle que `add2_ref`, qui lit son entrée tous les trois instants et prend trois instants pour calculer sa sortie. On cherche alors à le lancer en parallèle avec `add2_ref`, puis à additionner leurs résultats :

```
node add23_main(a:int) returns (o:int);
var c1,c2:bool; o1:int when c1; o2:int when c2;
let
  c1 = true -> not (pre c1);
  c2 = true -> pre (false -> not pre c2);
  o1 = add2_ref(c1, a when c1);
  o2 = add3_ref(c2, a when c2);
  o = current (o1 + o2);
tel
```

Le compilateur LUSTRE va rejeter ce programme, car on cherche à additionner deux flots `o1` et `o2` avec des horloges différentes, ce que le calcul d'horloges est fait pour rejeter :

```
PolluxError 997 in New::NewClock:
 (case 2) Invalid clock combination in equation of o in node add23_main
```

La situation est un peu différente en SIGNAL. Le calcul d'horloges ne rejette pas le programme, mais le compilateur ne génère pas non plus de code séquentiel. En effet, le compilateur trouve que le programme a deux horloges rapides, a priori sans liens. On peut voir l'arbre d'horloges correspondant à ce cas sur la figure 5.5b, qui est tirée de [GG10]. Cet arbre est en fait dans l'ordre

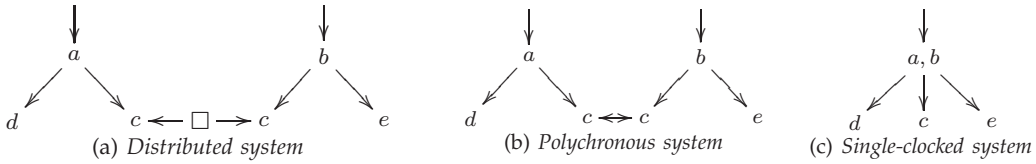


FIGURE 5.5 – Suréchantillonnage en SIGNAL

inverse de celui de REACTIVEML, puisque les racines sont les horloges les plus rapides. On peut alors générer un programme suivant deux techniques :

- La première consiste à générer un seul programme, dont l'horloge la plus rapide est l'union des deux horloges trouvées par le compilateur (figure 5.5c). Ce programme aura deux entrées supplémentaires, une par horloge rapide. Cette entrée devra être mise à vrai pour exécuter un pas de l'horloge correspondante. Le compilateur ajoute également des assertions qui vérifient les contraintes d'horloges qui n'ont pu être garanties à la compilation. Dans notre cas, il s'agit du fait que `o1` et `o2` sont synchrones. C'est donc à l'utilisateur du programme de donner les bonnes valeurs aux entrées correspondant aux deux horloges pour s'assurer que ces assertions sont vérifiées. C'est possible dans ce cas puisque le nombre d'instants locaux est connu statiquement, mais ce n'est pas forcément le cas de tous les programmes.
- La seconde solution consiste à générer deux programmes séparés, un par horloge rapide, lancés dans deux threads différents et communiquant par une FIFO (figure 5.5a) [GG10].

Le suréchantillonnage ne permet donc pas de répondre vraiment à la question du raffinement temporel, puisqu'il s'agit d'une opération non modulaire. Il faut en effet ralentir tous les autres processus pour obtenir un rythme plus rapide. On ne masque donc pas vraiment les instants rapides, ce qui est selon nous nécessaire pour le raffinement temporel et qui constitue une des composantes essentielles des domaines réactifs.

Troisième partie

Systèmes de types et analyses statiques

CHAPITRE 6

Calcul d'horloges

Dans ce chapitre, nous allons présenter une extension du système de types de REACTIVEML que nous appellerons *calcul d'horloges*. Ce travail a été publié dans [MPP13]. Son but est de garantir l'absence d'erreurs à l'exécution, en particulier que l'on n'accède pas à un signal en dehors de son domaine réactif et que l'on ne dépend jamais instantanément d'un signal lent. Nous l'appelons ainsi en référence au calcul d'horloges des langages synchrones flot de données [Cas92]. Nous verrons en effet que notre système de types présente de grandes similitudes avec le calcul d'horloges de LUCID SYNCHRONE [CP03] qui est aussi un système de types a la ML.

Nous allons tout d'abord présenter un rapide aperçu des systèmes de *types-et-effets*, dont fait partie le calcul d'horloges. Nous définirons ensuite une analyse très simple de bonne formation des expressions, qui assure la distinction syntaxique entre fonctions et processus. La partie suivante présentera formellement le calcul d'horloges et l'illustrera sur plusieurs exemples. Enfin, nous montrerons sa sûreté, c'est-à-dire suivant les mots célèbres de Milner [Mil78] : « well-typed programs cannot "go wrong" ». On prouvera donc que si un programme est bien typé, alors il se réduit en une valeur ou bien boucle indéfiniment dans la sémantique opérationnelle du chapitre 5.

Le système de types que nous présentons dans ce chapitre ne permet pas d'accepter autant de programmes que nous le souhaiterions, comme nous le verrons dans la dernière partie. Il permet cependant d'introduire les principes de l'analyse dans un cadre plus simple. Nous l'étendrons dans une version plus utile dans le chapitre 8, pour laquelle la preuve de sûreté sera toujours valable.

6.1 Introduction sur les systèmes de types-et-effets

Les systèmes de types-et-effets tirent leur origine de la présence de traits impératifs tels que les références dans les langages fonctionnels typés. Ils permettent en effet d'éviter les problèmes présents dans les premières versions de ML et liés à la généralisation des types des références. Ils ont ensuite été remplacés pour cette utilisation par un critère syntaxique pour limiter la généralisation [Tof90] bien plus simple à mettre en œuvre, qui est par exemple utilisé dans OCAML et REACTIVEML comme nous le verrons un peu plus loin.

L'idée des systèmes de types-et-effets [LG88] est d'associer à chaque expression un type, qui décrit le résultat de l'expression, mais aussi un *effet* qui rend compte de l'effet de l'évaluation de l'expression. Ces systèmes ont notamment été utilisés pour la gestion de la mémoire avec des régions [TJ92, TT97]. Il s'agit d'une alternative à la gestion automatique de la mémoire par un *garbage collector*. Les références sont allouées dans des régions et désallouées à la fin de l'exécution de la région. L'effet d'une expression est alors l'ensemble des noms des régions des références lues ou écrites par l'expression. L'inférence dans les systèmes de types-et-effets [JG91, TB98] est une extension de l'inférence de ML à la Damas-Milner [DM82]. Les preuves de sûreté [CHT02] reprennent également les principes des preuves syntaxiques usuelles [Pie02].

Les systèmes de types-et-effets ont ensuite été utilisés pour de nombreuses applications, allant de la gestion des exceptions à la sécurité (voir [MM09] pour plusieurs exemples). Le chapitre 5 de [NN99] et le chapitre 3 de [Pie05] donnent un aperçu global des systèmes de types-et-effets

$$\frac{}{k \vdash x} \qquad \frac{}{k \vdash c} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash (e_1, e_2)} \qquad (\text{ABS})\ \frac{0 \vdash e_1}{k \vdash \lambda x.e_1} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash e_1\ e_2} \qquad \frac{0 \vdash e_1}{k \vdash \texttt{rec}\ x = e_1}$$

$$(\text{PROCABS})\ \frac{1 \vdash e_1}{k \vdash \texttt{process}\ e_1} \qquad \frac{0 \vdash e_1}{1 \vdash \texttt{run}\ e_1} \qquad \frac{k \vdash e_1 \qquad k \vdash e_2 \qquad k \vdash e_3}{k \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3}$$

$$(\text{SIGNAL})\ \frac{0 \vdash e_h \qquad 0 \vdash e_{ck} \qquad 0 \vdash e_d \qquad 0 \vdash e_g \qquad 0 \vdash e_{rck} \qquad k \vdash e_1}{k \vdash \texttt{signal}\ x\ (\texttt{h} : e_h, \texttt{ck} : e_{ck}, \texttt{d} : e_d, \texttt{g} : e_g, \texttt{rck} : e_{rck})\ \texttt{in}\ e_1} \qquad \frac{0 \vdash e_s \qquad 0 \vdash e_1}{k \vdash \texttt{emit}\ e_s\ e_1}$$

$$\frac{0 \vdash e_s}{k \vdash \texttt{last}\ e_s} \qquad \frac{0 \vdash e_s \qquad 1 \vdash e_1 \qquad 1 \vdash e_2}{1 \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \qquad \frac{1 \vdash e_1 \qquad 0 \vdash e_s \qquad 1 \vdash e_2}{1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2}$$

$$\frac{1 \vdash e_1 \qquad 0 \vdash e_s}{1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e_s} \qquad \frac{1 \vdash e_1 \qquad 0 \vdash e_b}{1 \vdash \texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1} \qquad \frac{0 \vdash e_{ck}}{1 \vdash \texttt{pause}\ e_{ck}} \qquad \frac{0 \vdash e_{ck}}{1 \vdash \texttt{qpause}\ e_{ck}}$$

$$k \vdash \texttt{local\_ck} \qquad \frac{1 \vdash e_1}{1 \vdash e_1\ \texttt{in}\ (ck', i/j)}$$

FIGURE 6.1 – Bonne formation des expressions

et de leurs applications. On peut citer en particulier une analyse des exceptions levées par une fonction en OCAML [LP00]. Une autre application des systèmes de types-et-effets consiste à voir les effets non pas comme un ensemble, mais comme une abstraction de la structure du programme. Il peut s'agir par exemple d'une trace du programme [SSVH08] ou encore d'une abstraction du programme dans une algèbre de processus [NN93, ANN99]. C'est l'approche que nous utiliserons pour l'analyse de réactivité du chapitre 7.

6.2 Bonne formation des expressions

Le langage REACTIVEML impose une séparation stricte et syntaxique entre les expressions instantanées et les processus. Pour cela, on définit un prédicat de bonne formation des expressions que l'on note $k \vdash e$, où $k \in \{0, 1\}$ (voir [MP08b]). Le prédicat $0 \vdash e$ signifie que e est une expression instantanée (on dit aussi combinatoire). $1 \vdash e$ signifie que e est une expression réactive (on dit aussi séquentielle ou à mémoire en reprenant la terminologie des circuits numériques synchrones). Nous rappelons ici la définition de ce prédicat, que nous étendons au passage à notre extension du langage, puisque nous l'utiliserons par la suite pour nos preuves de sûreté.

Les règles définissant ce prédicat dans le langage étendu sont données sur la figure 6.1. Il n'y a aucune nouveauté par rapport à la définition de ce prédicat en REACTIVEML. Nous avons simplement ajouté les règles correspondant aux nouvelles instructions du langage. Nous ne discuterons pas des choix faits dans la définition de ce prédicat, puisque cela est déjà fait dans [Man06]. Nous pouvons tout de même rappeler les points les plus importants :

- $k \vdash e$ signifie que $0 \vdash e$ et $1 \vdash e$. L'expression peut donc être utilisée dans n'importe quel contexte, aussi bien dans une expression instantanée que dans une expression réactive. C'est par exemple le cas des variables, des constantes ou encore de l'application.
- Le corps d'une fonction doit être une expression instantanée (ABS), alors que celui d'un processus peut être réactif (PROCABS).

– Nous avons vu pendant la définition de la sémantique comportementale que nous souhaitions être plus strict que la première version de REACTIVEML et interdire l'émission de signal dans certaines expressions, comme par exemple les arguments d'une fonction. Nous avons donc supposé que l'événement associé à ces expressions était vide. On peut remarquer que l'analyse de bonne formation que nous présentons ici ne garantit pas cette propriété. C'est le calcul d'horloges qui s'occupera de cette tâche. On pourrait bien entendu modifier l'analyse de bonne formation pour ne permettre les émissions que dans les processus, mais cela nous parait trop contraignant.
– Nous verrons dans le chapitre 9 que, dans la version distribuée du moteur d'exécution de REACTIVEML, la création de signal dans une expression instantanée est très inefficace. On pourrait donc l'interdire en changeant la conclusion de la règle SIGNAL. Il suffit de remplacer le k de la conclusion par 1 pour forcer les signaux à être déclarés dans un processus. Nous nous sommes restreint à ce cas dans les exemples de ce manuscrit.

6.3 Système de types

Les domaines réactifs induisent plusieurs limites sur l'utilisation des signaux. La première est que l'on ne peut pas lire ou émettre sur un signal en dehors du domaine réactif auquel il est attaché. En effet, on ne peut pas voir les variations de sa valeur, puisque le domaine réactif masque ses instants locaux. Le rôle du calcul d'horloges est de garantir qu'un signal n'échappe jamais de son domaine. Ainsi, il ne doit pas apparaître dans la valeur de retour du domaine, comme on l'a déjà vu avec le processus `result_escape` (page 35) :

exemples/ch3/result_escape.rml
```
let process result_escape =
  domain ck by 10 do
    signal s in s
  done
```

Puisqu'un signal est une valeur de première classe, il peut également échapper de son domaine en étant émis sur un signal lent, comme dans l'exemple du processus `signal_escape` que le calcul d'horloges doit aussi rejeter :

exemples/ch3/signal_escape.rml
```
let process signal_escape =
  signal slow in
  domain ck by 10 do
    signal fast default 0 gather (+) in
    emit slow fast
  done
```

Enfin, le dernier cas à considérer est celui où l'accès au signal est masqué par une fonction :

exemples/ch3/effect_escape.rml
```
let process effect_escape =
  domain ck by 10 do
    signal fast in
    let f () = emit fast in f
  done
```

La notion d'effet permettra de rejeter ce programme, puisque l'effet d'une fonction contiendra toutes les horloges des signaux auxquels elle accède.

La seconde limite est que l'on ne peut pas dépendre instantanément d'un signal lent. Cette restriction est nécessaire pour ne pas faire d'hypothèses contradictoires sur la présence d'un signal au cours du même instant, comme nous l'avons vu avec le processus `immediate_dep_wrong` (page 36) :

exemples/ch3/immediate_dep_wrong.rml

```
let process immediate_dep_wrong =
  signal s in
  domain ck by 2 do
    await immediate s; print_string "Ok"
    ||
    pause ck; emit s
  done
```

Pendant le premier instant de l'horloge `ck`, on suppose que le signal `s` est absent. La première branche du parallèle est donc en attente, alors que la seconde attend l'instant suivant de `ck`. Au second instant de `ck`, le signal `s` devient présent. On réveille donc la première branche et on affiche le message `"Ok"`. Nous souhaitons rejeter ce programme car il fait deux hypothèses différentes sur la présence du signal `s` au cours du même instant, ce qui est contraire au principe du synchrone.

Il faut donc garantir que les dépendances immédiates comme `present` ou `do/when` ne sont faites que pour des signaux sur l'horloge locale. Nous verrons dans le chapitre 10 que cette restriction peut être levée dans le cas où l'on ajoute les domaines réactifs dans le cadre plus statique d'ESTEREL, au prix de potentiels problèmes de causalité.

Types et notations

Les types du calcul d'horloges sont définis par :

$$
\begin{array}{lll}
ct ::= T \mid \alpha \mid \{ce\} \mid ct \times ct \mid (ct, ct)\ \texttt{event}\{ce|\} & & \text{(types)} \\
\qquad \mid ct \xrightarrow{cf} ct \mid ct\ \texttt{process}\{ce|cf\} & & \\
ce ::= \gamma \mid ck & & \text{(horloges)} \\
cf ::= \phi \mid \emptyset \mid \{ce\} \mid cf \cup cf & & \text{(effets)} \\
cs ::= ct \mid \forall\alpha.\ cs \mid \forall\gamma.\ cs \mid \forall\phi.\ cs & & \text{(schémas de type)} \\
\Gamma ::= \{x_1 \mapsto cs_1; \ldots; x_p \mapsto cs_p\} & & \text{(environnements)}
\end{array}
$$

Un type ct est soit un type de base T (comme `unit` ou `int`), une variable de type α, un type singleton $\{ce\}$, un produit $ct_1 \times ct_2$, un signal $(ct_1, ct_2)\ \texttt{event}\{ce|\}$, une fonction $ct_1 \xrightarrow{cf} ct_2$ ou un processus $ct\ \texttt{process}\{ce|cf\}$. Le type singleton $\{ce\}$ est le type de l'horloge ce, qui est soit une variable d'horloge γ, soit un nom d'horloge ck. Le type $(ct_1, ct_2)\ \texttt{event}\{ce|\}$ d'un signal est paramétré par le type ct_1 des valeurs émises, le type ct_2 de la valeur reçue (et de sa valeur par défaut) et son horloge ce. Le type $ct_1 \xrightarrow{cf} ct_2$ d'une fonction est paramétré par le type ct_1 de son argument, son type de retour ct_2 et son effet cf. Un effet cf est un ensemble d'horloges ou une variable d'effet ϕ. Le type $ct\ \texttt{process}\{ce|cf\}$ d'un processus est caractérisé par son type de retour ct, son horloge d'activation ce et son effet cf. Un environnement de typage Γ est une fonction partielle des variables dans les schémas de type cs.

Les schémas de type cs généralisent les trois sortes de variables. L'instanciation d'un schéma de type, notée $ct \leq cs$, et la généralisation d'un type ct dans un environnement de typage Γ, notée $gen(ct, e, \Gamma)$, sont définies de façon classique [Pie02] par :

$$
cs[\alpha \leftarrow ct] \leq \forall\alpha.\ cs \qquad cs[\gamma \leftarrow ck] \leq \forall\gamma.\ cs \qquad cs[\phi \leftarrow cf] \leq \forall\phi.\ cs
$$

$$
gen(ct, e, \Gamma) = \begin{cases} ct & \text{si } e \text{ est expansive} \\ \forall\bar{\alpha}.\forall\bar{\gamma}.\forall\bar{\phi}.\ ct & \text{sinon, avec } \bar{\alpha}, \bar{\gamma}, \bar{\phi} = ftv(ct) \setminus ftv(\Gamma) \end{cases}
$$

où $ftv(ct)$ est l'ensemble des variables de type, d'horloge et d'effet libres dans ct. Puisque les signaux sont des structures mutables, on doit prendre garde à ne pas généraliser les expressions qui allouent des signaux. On distingue pour cela les expressions *expansives* [Tof90] qui peuvent allouer de nouveaux signaux et pour lesquelles les types ne sont pas généralisés. Les expressions

non-expansives e_{ne} sont définies par :

$$\begin{aligned} e_{ne} ::= {} & x \mid c \mid (e_{ne}, e_{ne}) \mid \lambda x.e \mid \texttt{process}\ e \\ & \mid \texttt{let}\ x = e_{ne}\ \texttt{and}\ x = e_{ne}\ \texttt{in}\ e_{ne} \mid \texttt{emit}\ e_{ne}\ e_{ne} \mid \texttt{last}\ e_{ne} \\ & \mid \texttt{present}\ e_{ne}\ \texttt{then}\ e_{ne}\ \texttt{else}\ e_{ne} \mid \texttt{do}\ e_{ne}\ \texttt{until}\ e_{ne}(x) \to e_{ne} \mid \texttt{do}\ e_{ne}\ \texttt{when}\ e_{ne} \\ & \mid \texttt{domain}\ x\ \texttt{by}\ e_{ne}\ \texttt{do}\ e_{ne} \mid \texttt{pause}\ e_{ne} \mid \texttt{qpause}\ e_{ne} \mid \texttt{local\_ck} \end{aligned}$$

Règles

Le système de types est défini par un jugement de la forme :

$$\Gamma, ce \vdash e : ct \mid cf$$

qui signifie que dans l'environnement de typage Γ et l'horloge locale ce, l'expression e a le type ct et l'effet cf. Les programmes sont typés dans un environnement de typage initial Γ_0 donnant le type des différentes primitives et défini par :

$$\begin{aligned} \Gamma_0 \triangleq [\, & \mathit{global\_ck} : \{\top_{ck}\};\ \texttt{pause} : \forall\gamma.\,\{\gamma\} \xrightarrow{\{\gamma\}} \texttt{unit};\ \texttt{qpause} : \forall\gamma.\,\{\gamma\} \xrightarrow{\{\gamma\}} \texttt{unit}; \\ & \texttt{last} : \forall\alpha_1,\alpha_2,\gamma.\,(\alpha_1,\alpha_2)\ \texttt{event}\{\gamma|\} \xrightarrow{\{\gamma\}} \alpha_2; \\ & \texttt{emit} : \forall\alpha_1,\alpha_2,\gamma.\,(\alpha_1,\alpha_2)\ \texttt{event}\{\gamma|\} \xrightarrow{\emptyset} \alpha_1 \xrightarrow{\{\gamma\}} \texttt{unit}; \\ & \texttt{true} : \texttt{bool};\ \texttt{fst} : \forall\alpha_1,\alpha_2.\,\alpha_1 \times \alpha_2 \xrightarrow{\emptyset} \alpha_1; \ldots] \end{aligned}$$

Les opérateurs `pause` et `qpause` prennent en argument une horloge de type $\{\gamma\}$ et ont un effet uniquement sur cette horloge. Les opérateurs `last` et `emit` ont un effet sur l'horloge γ du signal pris en argument.

La définition du système de types est donnée sur la figure 6.2 :

- On utilise la présentation dirigée par la syntaxe habituelle de ML : l'instanciation se fait au niveau des variables (VAR et CONST), alors que la généralisation se fait pour le `let`.
- L'expression `local_ck` a un type singleton $\{ce\}$, où ce est l'horloge locale que l'on trouve dans le jugement de typage à droite de l'environnement de typage.
- Les règles de l'abstraction et de l'application sont les règles classiques dans un système de types-et-effets. L'idée est de stocker dans le type de la fonction l'effet cf_1 de son corps. La fonction elle-même a alors un effet vide, comme toute valeur. On extrait cet effet au moment de l'application : l'effet de l'application d'une fonction est l'effet du corps de cette fonction.
- On procède de la même façon pour les processus. Pour typer le corps e_1 d'un processus, on utilise son horloge d'activation ce' comme nouvelle horloge locale. Au cours de l'inférence, il s'agira d'une variable fraiche. On ne peut lancer que des processus dont l'horloge d'activation est égale à l'horloge locale ce. Cela se fera typiquement en instanciant un processus dont l'horloge d'activation est une variable d'horloge quantifiée universellement.
- Dans le cas du `let`/`and`/`in`, on calcule d'abord le type et l'effet des deux branches. On type ensuite e_3 après avoir généralisé les types de e_1 et e_2. L'effet de l'expression est obtenu en additionnant les effets des trois sous-expressions.
- L'horloge d'un signal est obtenue en typant l'expression e_{ck} qui doit être une horloge de type singleton $\{ck'\}$. On peut remarquer que le système de types n'impose aucune condition sur l'horloge ck'' de réinitialisation du signal. Elle apparait simplement dans l'effet de l'expression, ce qui garantit qu'il s'agit d'une horloge plus lente que l'horloge locale. On note également que la fonction de combinaison du signal, qui est donnée par l'expression e_g, doit être une fonction combinatoire sans aucun effet. Ceci s'explique par le fait que cette fonction est appelée au cours de la fin de l'instant, pendant laquelle il est impossible d'émettre un signal.
- Dans le cas du `present`, on empêche la dépendance immédiate sur un signal lent en forçant l'horloge du signal e_s à être égale à l'horloge locale ce. On ajoute également l'horloge locale

$$(\textsc{Var})\frac{ct \leq \Gamma(x)}{\Gamma, ce \vdash x : ct \mid \emptyset} \qquad (\textsc{Const})\frac{ct \leq \Gamma_0(c)}{\Gamma, ce \vdash c : ct \mid \emptyset} \qquad \frac{}{\Gamma, ce \vdash \texttt{local\_ck} : \{ce\} \mid \emptyset}$$

$$\frac{\Gamma, ce \vdash e_1 : ct_1 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_2 \mid \emptyset}{\Gamma, ce \vdash (e_1, e_2) : ct_1 \times ct_2 \mid \emptyset} \qquad \frac{\Gamma[x \mapsto ct], ce \vdash e_1 : ct \mid cf}{\Gamma, ce \vdash \texttt{rec}\ x = e_1 : ct \mid cf}$$

$$\frac{\Gamma[x \mapsto ct_2], ce \vdash e_1 : ct_1 \mid cf_1}{\Gamma, ce \vdash \lambda x.e_1 : ct_2 \xrightarrow{cf_1} ct_1 \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_1 : ct_2 \xrightarrow{cf} ct_1 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_2 \mid \emptyset}{\Gamma, ce \vdash e_1\ e_2 : ct_1 \mid cf}$$

$$\frac{\Gamma, ce' \vdash e_1 : ct \mid cf_1}{\Gamma, ce \vdash \texttt{process}\ e_1 : ct\ \texttt{process}\{ce'|cf_1\} \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_p : ct\ \texttt{process}\{ce|cf\} \mid \emptyset}{\Gamma, ce \vdash \texttt{run}\ e_p : ct \mid cf}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 : ct_1 \mid cf_1 \qquad \Gamma, ce \vdash e_2 : ct_2 \mid cf_2 \\ \Gamma[x_1 \mapsto gen(ct_1, e_1, \Gamma); x_2 \mapsto gen(ct_2, e_2, \Gamma)], ce \vdash e_3 : ct \mid cf_3\end{array}}{\Gamma, ce \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3 : ct \mid cf_1 \cup cf_2 \cup cf_3}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_{ck} : \{ce'\} \mid \emptyset \qquad \Gamma, ce \vdash e_d : ct_2 \mid \emptyset \qquad \Gamma, ce \vdash e_g : ct_1 \xrightarrow{\emptyset} ct_2 \xrightarrow{\emptyset} ct_2 \mid \emptyset \\ \Gamma, ce \vdash e_{rck} : \{ce''\} \mid \emptyset \qquad \Gamma[x \mapsto (ct_1, ct_2)\ \texttt{event}\{ce'|\}], ce \vdash e_1 : ct \mid cf_1\end{array}}{\Gamma, ce \vdash \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e_{ck}, \texttt{d} : e_d, \texttt{g} : e_g, \texttt{rck} : e_{rck})\ \texttt{in}\ e_1 : ct \mid cf_1 \cup \{ce'\} \cup \{ce''\}}$$

$$\frac{\Gamma, ce \vdash e_s : (ct_1, ct_2)\ \texttt{event}\{ce|\} \mid \emptyset \qquad \Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_2 : ct \mid cf_2}{\Gamma, ce \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : ct \mid cf_1 \cup cf_2 \cup \{ce\}}$$

$$\frac{\Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_s : (ct_1, ct_2)\ \texttt{event}\{ce'|\} \mid \emptyset \qquad \Gamma[x \mapsto ct_2], ce \vdash e_2 : ct \mid cf_2}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \rightarrow e_2 : ct \mid cf_1 \cup cf_2 \cup \{ce'\}}$$

$$\frac{\Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_s : (ct_1, ct_2)\ \texttt{event}\{ce|\} \mid \emptyset}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{when}\ e_s : ct \mid cf_1 \cup \{ce\}}$$

$$(\textsc{Domain})\frac{\Gamma, ce \vdash e_b : \texttt{int} \mid \emptyset \qquad \Gamma[x \mapsto \{\gamma\}], \gamma \vdash e_1 : ct \mid cf_1 \qquad \gamma \notin ftv(\Gamma, ct)}{\Gamma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1 : ct \mid cf_1 \setminus \{\gamma\}}$$

$$(\textsc{In})\frac{\Gamma, ck' \vdash e_1 : ct \mid cf_1}{\Gamma, ce \vdash e_1\ \texttt{in}\ (ck', i/j) : ct \mid cf_1 \setminus \{ck'\}}$$

FIGURE 6.2 – Calcul d'horloges

dans l'effet de l'expression, en plus des effets cf_1 et cf_2 des deux branches. On procède de la même façon pour le `do/when`. C'est pour gérer le cas des dépendances immédiates que les processus doivent avoir une horloge d'activation.
- Dans la règle du `do/until`, il faut garantir que l'on accède à un signal dont l'horloge est plus lente ou égale à l'horloge locale. On ajoute pour cela l'horloge ce' du signal à l'effet de l'expression. C'est ensuite la règle DOMAIN qui garantit que cette horloge est bien accessible.
- La règle la plus importante de ce système de types-et-effets est bien entendu celle des domaines réactifs (DOMAIN). Elle garantit que le nom de l'horloge locale ne s'échappe pas du corps du domaine. Pour cela, on créé une variable d'horloge γ fraiche, que l'on utilise comme horloge locale pour typer le corps e_1 du domaine. La variable x représentant cette horloge a donc un type singleton $\{\gamma\}$. On vérifie ensuite que γ n'apparait ni dans le type ct du corps, ni dans le type d'une variable de l'environnement de typage Γ. On reprend ici le principe des types abstraits de [LO92], qui est aussi utilisé pour le typage des horloges en LUCID SYNCHRONE [CP03]. L'effet du domaine est obtenu en enlevant l'horloge locale γ de l'effet cf_1 du corps.
- On procède de la même façon pour un domaine en cours d'exécution e_1 `in` $(ck', i/j)$ (IN). On utilise alors l'horloge locale ck' comme horloge locale pour typer le corps. On obtient l'effet du domaine en enlevant cette horloge de l'effet cf_1 du corps.
- On remarque que de nombreuses règles imposent un effet vide pour certaines sous-expressions (par exemple pour les arguments de l'application ou de la déclaration d'un signal ou encore les éléments d'une paire). Il s'agit des sous-expressions qui doivent être des expressions ML sans effet, pour lesquelles on a supposé dans la sémantique comportementale qu'elles ne pouvaient pas émettre de valeurs sur un signal. L'analyse de bonne formation des expressions que nous avons présentée dans la partie 6.2 assure déjà que ces expressions sont instantanées (c'est-à-dire qu'elles vérifient $0 \vdash e$).
  Dans la plupart des cas, ce choix d'imposer l'absence d'effets permet juste de simplifier le système de types et la sémantique. Il ne réduit pas l'expressivité du langage, puisque l'on peut toujours définir préalablement les expressions avec effets en utilisant un `let`. Il y a un seul cas où il s'agit d'une question de sûreté : pour la fonction de combinaison d'un signal. En effet, l'émission d'un signal dans cette fonction n'a aucun sens, ni même la lecture de la dernière valeur d'un signal. On peut voir d'ailleurs que la sémantique considère cette fonction comme « pure » et ne décrit pas son évaluation. On pourrait la définir en prenant la sémantique de ML, qui ne mentionne pas l'environnement de signaux, et en manipulant les processus et signaux comme des types abstraits. Il faut donc que cette évaluation ne dépende pas de l'environnement de signaux, ce que l'on garantit en s'assurant que l'effet de la fonction est vide.

Propriété

Le calcul d'horloges est une extension du système de types de REACTIVEML [MP08b], que l'on peut aisément étendre aux nouvelles constructions du langage. Cela signifie que si un programme est bien typé dans le calcul d'horloges, alors il est aussi bien typé dans le système de types de REACTIVEML et il a le « même » type. La réciproque n'est bien entendu pas vraie, puisque le but du calcul d'horloges est justement de rejeter plus de programmes. Un programme REACTIVEML bien typé n'utilisant pas de domaines réactifs n'a jamais de problèmes d'échappement de portée, mais il pourra quand même être rejeté à cause des limitations du système de types que nous expliquerons dans la partie 6.5. Nous verrons comment lever ces restrictions dans le chapitre 8.

Plus formellement, on peut définir une opération d'effacement, que l'on note $|ct|_\tau$, qui associe à un type du calcul d'horloges ct un type normal τ. Cette opération est définie par :

$$|T|_\tau \triangleq T$$
$$|\alpha|_\tau \triangleq \alpha$$
$$|\{ce\}|_\tau \triangleq \texttt{clock}$$

$$|ct_1 \times ct_2|_\tau \triangleq |ct_1|_\tau \times |ct_2|_\tau$$
$$|(ct_1, ct_2)\ \texttt{event}\{ce|\}|_\tau \triangleq (|ct_1|_\tau, |ct_2|_\tau)\ \texttt{event}$$
$$|ct_2 \xrightarrow{cf} ct_1|_\tau \triangleq |ct_2|_\tau \to |ct_1|_\tau$$
$$|ct_1\ \texttt{process}\{ce|cf\}|_\tau \triangleq |ct_1|_\tau\ \texttt{process}$$
$$|\forall\alpha.\ ct|_\tau \triangleq \forall\alpha.\ |ct|_\tau$$
$$|\forall\gamma.\ ct|_\tau \triangleq |\forall\phi.\ ct|_\tau \triangleq |ct|_\tau$$

On étend cette opération aux environnements de typage. On peut ensuite énoncer la propriété suivante, où l'on note $\vdash_\tau$ les jugements du système de types de REACTIVEML [MP08b] :

Propriété 6.1. *Si* $\Gamma, ce \vdash e : ct \mid cf$*, alors* $|\Gamma|_\tau \vdash_\tau e : |ct|_\tau$.

Démonstration. Par induction sur les règles de typage. □

Syntaxe concrète des types

La forme générale d'un type est maintenant :

```
(ct1, .., ctn) ident{ c1, .., cm | e1, .., ep }
```

Il est donc paramétré par une liste de types `ct`, une liste d'horloges `c` et une liste d'effets `e`. Le type d'un signal est ainsi `(t1, t2) event{c|}`, où `t1` est le type des valeurs émises, `t2` est le type de la valeur reçue et `c` est son horloge. Celui d'un processus est `t process{c|e}`, où `t` est son type de retour, `c` son horloge d'activation et `e` son effet. On note `'c` les variables d'horloge et `?ck` la variable d'horloge associée au domaine réactif d'horloge `ck`, que l'on a notée γ dans les règles du système de types. On note `''e` les variables d'effet et + l'union des effets. Enfin, on note `[c]` le type singleton associé à l'horloge `c`.

Le type des fonctions est noté `ct1 =>{eff} ct2`. On écrit `a -> b` comme raccourci pour `a =>{0} b`, c'est-à-dire pour désigner une fonction sans effet. C'est donc le type que l'on donne aux fonctions importées de la librairie standard d'OCAML. Enfin, on peut aussi écrire `a => b` pour `a =>{''f} b` où `''f` est une variable d'effet fraiche pour l'ensemble du programme. Il s'agit donc d'une fonction avec un effet quelconque. On peut, de la même façon, omettre l'horloge locale et l'effet d'un processus, qui sont alors remplis avec des variables fraiches.

Pour simplifier l'écriture des types et des effets, on ajoute une nouvelle passe au compilateur. On a vu, en effet, que l'on pouvait omettre l'effet d'une fonction ou d'un processus. Le but de cette passe est d'ajouter les variables fraiches correspondantes dans les paramètres de la déclaration du type. Par exemple, les définitions suivantes, valides en OCAML et REACTIVEML :

```
type 'a t = 'a => unit
type t2 = { l1 : int t; l2 : float t }
```

sont traduites en :

```
type 'a t{|''e0} = 'a =>{''e0} unit
type t2{|''e1, ''e2} = { l1 : int t{|''e1}; l2 : float t{|''e2} }
```

Cette passe gère également le cas des types récursifs. La seule chose que l'on puisse faire dans ce cas est de supposer que toutes les occurrences du même type récursif sont utilisées avec les mêmes variables. Par exemple, on transforme :

```
type t =
| Empty | Node of int => int * t * t
```

en

```
type t{|''e0} =
| Empty | Node of int =>{|''e0} int * t{|''e0} * t{|''e0}
```

Ce mécanisme s'étend de façon similaire à des types mutuellement récursifs.

Implémentation

L'inférence des types et effets des expressions ne pose aucun problème particulier. On utilise le même algorithme d'unification que pour REACTIVEML, auquel on ajoute le calcul des effets. On doit également ajouter l'horloge locale en argument de la fonction d'inférence. On décrira l'algorithme de calcul des effets de façon plus formelle dans le chapitre 8. L'autre nouveauté est le test d'échappement de la règle DOMAIN. Le test d'appartenance d'une variable à l'environnement ne pose pas de difficulté puisque l'on doit déjà le faire au moment de la généralisation. L'implémentation s'inspire de celle de LUCID SYNCHRONE [CP03], elle-même tirée de [MP94].

Exemples

On souhaite tout d'abord rejeter les programmes dans lesquels le type de retour d'un domaine réactif contient un signal rapide, comme l'exemple suivant (que l'on a déjà vu page 35) :

exemples/ch6/result_escape.rml
```
let process result_escape =
  domain ck do
    signal s in s
  done
```

Ce programme est rejeté par le compilateur avec le message d'erreur suivant :

```
The clock of this expression, that is,
('_a list , '_a) event{?ck|} process {'_c0|0} depends on ck which escapes its scope.
```

En effet, la variable fraiche γ associée à l'horloge locale apparait dans le type du corps du domaine, que l'on note $e \triangleq \texttt{signal}\ s\ \texttt{in}\ s$:

$$\frac{\Gamma_0, \gamma \vdash e : (\alpha, \alpha\ \texttt{list})\ \texttt{event}\{\gamma|\}\ |\ \{\gamma\}}{\Gamma_0, \top_{ck} \vdash \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e : (\alpha, \alpha\ \texttt{list})\ \texttt{event}\{\gamma|\}\ |\ \emptyset}$$

L'autre cas à rejeter est celui où un signal rapide échappe de son domaine en étant émis sur un signal lent :

exemples/ch6/signal_escape.rml
```
let process signal_escape =
  signal slow in
  domain ck do
    signal fast default 0 gather (+) in
    emit slow fast
  done
```

```
The emitted value has clock (int , int) event{?ck|}, and would thus escape its scope 'ck'.
```

Cette fois, le programme est rejeté car la variable γ apparait dans le type d'une variable de l'environnement, en l'occurrence la variable *slow* :

$$slow : ((\texttt{int}, \texttt{int})\ \texttt{event}\{\gamma|\}, (\texttt{int}, \texttt{int})\ \texttt{event}\{\gamma|\})\ \texttt{event}\{\top_{ck}|\}$$

L'utilisation d'effets dans le système de types permet également de traiter le cas où l'accès au signal est caché dans une fonction :

exemples/ch6/effect_escape.rml
```
let process effect_escape =
  domain ck do
    signal fast in
    (fun () -> emit fast)
  done
```

```
The clock of this expression, that is, (unit =>{?ck} unit) process {'_c0|0}
depends on ck which escapes its scope.
```

La variable γ apparait à nouveau dans le type du corps du domaine, c'est-à-dire $\texttt{unit} \xrightarrow{\{\gamma\}} \texttt{unit}$.

Le calcul d'horloges doit également rejeter les processus où le corps d'un domaine réactif dépend instantanément d'un signal lent :

exemples/ch6/immediate_dep_wrong.rml

```
let process immediate_dep_wrong =
  signal s in
  domain ck do
    present s then () else ()
  done
```

```
This immediate dependency would make 'ck' escape its scope.
```

La raison pour laquelle le système de types n'accepte pas ce processus est un peu subtile. Notons $e \triangleq$ `present` s `then` () `else` () et $\Gamma \triangleq \Gamma_0[s \mapsto (\texttt{unit}, \texttt{unit list})\ \texttt{event}\{\gamma|\}]$. On obtient alors la dérivation de type suivante :

$$\dfrac{\dfrac{\dfrac{\Gamma, \gamma \vdash s : (\texttt{unit}, \texttt{unit list})\ \texttt{event}\{\gamma|\} \mid \emptyset \qquad \Gamma, \gamma \vdash () : \texttt{unit} \mid \emptyset \qquad \Gamma, \gamma \vdash () : \texttt{unit} \mid \emptyset}{\Gamma, \gamma \vdash \texttt{present}\ s\ \texttt{then}\ ()\ \texttt{else}\ () : \texttt{unit} \mid \{\gamma\}}}{\Gamma, \gamma' \vdash \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e : \texttt{unit} \mid \emptyset}}{\Gamma_0, \gamma' \vdash \texttt{signal}\ s\ \texttt{in domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e : \texttt{unit} \mid \emptyset}$$

On calcule la dérivation dans l'horloge γ' d'activation du processus `immediate_dep_wrong`. On voit que le test de présence sur s force son horloge à être égale à l'horloge locale du domaine, c'est-à-dire γ. Le programme est ensuite rejeté car γ apparait libre dans le type de s, qui est une variable de l'environnement où l'on type le domaine.

On peut également s'intéresser au type de programmes acceptés. Un exemple intéressant est le processus switch (page 18) :

```
let process sustain s =
  loop emit s; pause end
```

```
val sustain : ('a , unit) event{'c0|} ->{0} unit process {'c1|'c0 + 'c1}
```

```
let process switch s_in s_out =
  loop
    do run sustain s_out until s_in done;
    await immediate s_in; pause
  end
```

```
val switch : ('a , 'b) event{'c0|} ->{0}
    ('c , unit) event{'c1|} ->{0} unit process {'c0|'c1 + 'c0}
```

On remarque en particulier que l'horloge du signal `s_in` doit être égale à l'horloge `'c0` d'activation du processus `switch` puisque le corps du processus dépend immédiatement de `s_in`. L'horloge `'c1` de `s_out` n'est pas contrainte et est donc égale à une autre variable. On peut aussi considérer le processus par_map (page 18) :

```
let rec process par_map p l =
  match l with
  | [] -> []
  | x :: l -> let x' = run p x
              and l' = run par_map p l in
              x' :: l'
```

```
val par_map : ('a ->{''e0} 'b process {'c0|''e1}) ->{0}
  'a list ->{0} 'b list process {'c0|''e0 + ''e1}
```

On voit que l'effet du processus `par_map` est la somme de l'effet `''e0` obtenu par l'application de `p` à la variable `x` et de l'effet `''e1` du processus résultant.

6.4 Preuve de sûreté

Nous montrons maintenant la sûreté du calcul d'horloges, c'est-à-dire que si un programme est bien typé, alors il se réduit en une valeur ou bien il peut se réduire indéfiniment. On utilisera pour cela des techniques classiques de preuves syntaxiques de systèmes de types [Pie02], en utilisant la sémantique opérationnelle du chapitre 5. La preuve se base sur deux lemmes principaux :

- La préservation du typage par les réductions (habituellement appelée *subject reduction*), qui signifie qu'une expression bien typée se réduit en un pas élémentaire en une expression de même type.
- La propriété de progrès, c'est-à-dire qu'une expression bien typée est soit une valeur, soit peut se réduire. Dans le cas de REACTIVEML, les choses sont légèrement différentes : on va montrer qu'une expression bien typée est soit une expression de fin d'instant, c'est-à-dire que l'on peut la réduire avec la réduction de fin d'instant, soit peut se réduire avec la réduction d'instant.

Pour traiter le cas des horloges, nous suivons la démarche de [CHT02], qui montre la sûreté d'un système de types-et-effets utilisant des régions pour la gestion de la mémoire. La particularité de cette preuve est que l'on doit légèrement modifier les règles du calcul d'horloges pour ajouter un nouvel environnement de typage noté Σ. Celui-ci contient le type des noms de signaux, alors que l'environnement de typage Γ contient le type des variables. La vérification d'échappement de portée dans la règle DOMAIN ne se fera que pour Γ, pas pour Σ. En effet, lorsque l'on sort d'un domaine réactif, les types des signaux du domaine sont toujours enregistrés dans Σ, mais ils ne doivent pas apparaître dans Γ puisque cela signifierait qu'ils pourraient être utilisés dans le reste du programme.

Notations

Comme dans la preuve de [CHT02], on introduit un nouvel environnement pour la preuve :

Définition 6. Un *environnement de typage de signal* Σ est une fonction partielle des noms d'horloges dans les fonctions partielles des noms de signaux n dans les types ct.

On ne reprend pas la même structure que pour les environnements de signaux, puisque tous les environnements de signaux associés à la même horloge ont le même type. Le domaine de l'environnement de typage de signal Σ est donc l'ensemble des noms d'horloges, pas celui des piles. Le type de l'environnement local de signaux $\mathcal{S}(s)$ est ainsi donné par $\Sigma(\texttt{top}(s))$. On ajoute l'environnement de typage de signal dans les jugements de typage, qui sont maintenant de la forme :

$$\Gamma, \Sigma, ce \vdash e : ct \mid cf$$

En REACTIVEML classique, le type des noms de signaux était donné par l'environnement de typage Γ. Dans notre cas, on ajoute une nouvelle règle qui va chercher le type dans l'environnement de typage de signal Σ :

$$\Gamma, \Sigma, ce \vdash n^{ck} : \Sigma(ck)(n) \mid \emptyset \qquad\qquad \Gamma, ce \vdash ck : \{ck\} \mid \emptyset$$

On a également ajouté une règle pour donner le type des noms d'horloges, puisque l'on a besoin de typer toutes les valeurs. La différence de traitement entre les deux environnements de typage se voit dans la règle de typage des domaines réactifs qui devient :

$$\text{(DOMAIN')}\ \frac{\Gamma, \Sigma, \gamma \vdash e_1 : ct \mid cf_1 \qquad \Gamma, \Sigma, ce \vdash e_b : \texttt{int} \mid \emptyset \qquad \gamma \notin ftv(\Gamma, ct)}{\Gamma, \Sigma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1 : ct \mid cf_1 \setminus \{\gamma\}}$$

Le point important de cette règle est que l'on vérifie que l'horloge locale γ n'apparait pas dans l'environnement de typage Γ, mais on ne fait aucune vérification pour l'environnement de typage de signal Σ.

On peut tout d'abord montrer plusieurs lemmes simples.

Lemme 6.2. *Une valeur bien typée a un effet vide. Autrement dit, si* $\Gamma, \Sigma, ce \vdash v : ct \mid cf$*, alors* $cf = \emptyset$*.*

Démonstration. On doit ici vérifier les différents cas possibles, en utilisant soit la règle pour les constantes c, soit la règle de l'abstraction, soit celle de la définition de processus, soit les deux règles que l'on vient de donner pour les noms de signaux et d'horloges. □

Lemme 6.3. *Une valeur a le même type quelle que soit l'horloge locale. Si* $\Gamma, \Sigma, ce \vdash v : ct \mid \emptyset$*, alors* $\Gamma, \Sigma, ce' \vdash v : ct \mid \emptyset$ *pour tout* ce'*.*

Démonstration. L'horloge d'activation n'est utilisée que pour typer le corps des processus. Le seul cas intéressant est donc pour $v = \texttt{process}\ e_1$. Dans ce cas, on utilise une variable fraiche pour typer le corps du processus, donc l'horloge ce n'est jamais utilisée. □

On va maintenant faire le lien entre les environnements de signaux et les environnements de typage de signal qui les décrivent :

Définition 7. On dit qu'un environnement de signaux $\mathcal{S}$ est bien typé dans l'environnement de typage de signal Σ, ce que l'on note $\Sigma \vdash \mathcal{S}$, si :

– Tous les noms de signaux présents dans $\mathcal{S}$ sont aussi dans Σ :

$$\forall s \in Dom(\mathcal{S}).\ Dom(\mathcal{S}(s)) \subseteq Dom(\Sigma(\texttt{top}(s)))$$

– Pour toute pile $s \in Dom(\mathcal{S})$ et tout signal $n \in Dom(\mathcal{S}(s))$, il existe ct_1, ct_2 et ck' tels que :

$$\begin{array}{l}
\Sigma(\texttt{top}(s))(n) = (ct_1, ct_2)\ \texttt{event}\{\texttt{top}(s)|\} \\
\Gamma_0, \Sigma, \top_{ck} \vdash \mathcal{S}^d(s)(n) : ct_2 \mid \emptyset \\
\Gamma_0, \Sigma, \top_{ck} \vdash \mathcal{S}^g(s)(n) : ct_1 \xrightarrow{\emptyset} ct_2 \xrightarrow{\emptyset} ct_2 \mid \emptyset \\
\Gamma_0, \Sigma, \top_{ck} \vdash \mathcal{S}^m(s)(n) : ct_1\ \texttt{multiset} \mid \emptyset \\
\Gamma_0, \Sigma, \top_{ck} \vdash \mathcal{S}^h(s)(n) : \texttt{bool} \mid \emptyset \\
\Gamma_0, \Sigma, \top_{ck} \vdash \mathcal{S}^{rck}(s)(n) : \{ck'\} \mid \emptyset
\end{array}$$

On peut remarquer que le jugement $\Sigma \vdash \mathcal{S}$ ne fait pas intervenir d'horloge locale puisque l'on doit typer uniquement des valeurs, pour lesquelles le lemme 6.3 montre que l'on peut utiliser n'importe quelle horloge. On peut maintenant montrer le lemme suivant, qui établit que l'opération $\texttt{next}_C(s)$, définie page 59 et qui calcule le successeur d'un environnement local de signaux, rend un environnement local de signaux bien typé :

Lemme 6.4. *Si* $\Sigma \vdash \mathcal{S}$ *et que* $\mathcal{S}' = \mathcal{S} \sqcup \{\texttt{snext}(s', \{ck'\}) \mapsto \texttt{next}_{\{ck'\}}(\mathcal{S}(s'))\}$ *(où* $s' \in Dom(\mathcal{S})$ *et* $ck' = \texttt{top}(s')$*), alors* $\Sigma \vdash \mathcal{S}'$*.*

Démonstration. Tous les champs des signaux sont conservés, sauf la dernière valeur $\mathcal{S}^l(s)(n)$ qui peut être remplacée soit par la valeur par défaut du signal $\mathcal{S}^d(s)(n)$, soit par la valeur du signal à l'instant courant $\mathcal{S}^v(s)(n)$. Le nouvel environnement local de signaux est donc bien typé puisque toutes ces valeurs ont le même type ct_2. □

Nous allons maintenant pouvoir définir le jugement de typage d'une configuration $e/\mathcal{S}$:

$$\frac{\Gamma_0, \Sigma, ce \vdash e : ct \mid cf \qquad \Sigma \vdash \mathcal{S} \qquad cf \subseteq Clocks(s)}{\Sigma, s \vdash e/\mathcal{S} : ct \mid cf}$$

Une configuration $e/\mathcal{S}$ est bien typée si l'expression e et l'environnement de signaux $\mathcal{S}$ sont bien typés dans l'environnement de typage Γ_0 et l'environnement de typage de signal Σ et si l'effet cf de l'expression est inclus dans l'ensemble $Clocks(s)$ des horloges accessibles, c'est-à-dire l'ensemble des horloges plus lentes que l'horloge locale $\texttt{top}(s)$.

Définition 8. On dit que $e/\mathcal{S}$ est une configuration de fin d'instant pour l'horloge ck s'il existe e' et $\mathcal{S}'$ tels que $s, \{ck\} \vdash e/\mathcal{S} \longrightarrow_{\text{eoi}} e'/\mathcal{S}'$ avec $ck = \texttt{top}(s)$.

Preuve

Nous allons maintenant énoncer plusieurs lemmes que l'on prouvera avec des techniques classiques (partie 9.3 de [Pie02]). Le premier montre que l'on peut étendre les environnements de typage et obtenir un jugement toujours correct :

Lemme 6.5. *Si* $\Gamma, \Sigma, ce \vdash e : ct \mid cf$ *et* $\Gamma \subseteq \Gamma'$ *et* $\Sigma \subseteq \Sigma'$*, alors* $\Gamma', \Sigma', ce \vdash e : ct \mid cf$.

Le second est le lemme de substitution : on peut substituer une variable par une valeur de même type et obtenir un terme du même type :

Lemme 6.6 (Substitution)**.** *Si* $\Gamma; x : \forall\bar{\alpha}.\forall\bar{\gamma}.\forall\bar{\phi}.\ ct', \Sigma, ce \vdash e : ct \mid cf$ *et* $\Gamma, \Sigma, ce \vdash v : ct' \mid \emptyset$ *alors* $\Gamma, \Sigma, ck \vdash e[x \leftarrow v] : ct \mid cf$.

On peut également montrer un lemme similaire, mais cette fois où l'on substitue un nom d'horloge à une variable d'horloge :

Lemme 6.7 (Substitution d'horloge)**.** *Soit* $\theta = [\gamma \leftarrow ck']$ *une substitution qui remplace une variable d'horloge* γ *par un nom d'horloge* ck'*. Si* $\Gamma, \Sigma, ce \vdash e : ct \mid cf$*, alors* $\theta(\Gamma), \theta(\Sigma), \theta(ce) \vdash e : \theta(ct) \mid \theta(cf)$.

Démonstration. Par induction sur les règles de typage. □

La première chose que l'on doit montrer est que la réduction de fin d'instant préserve le typage. On doit le faire en premier puisque la réduction d'instant utilise la réduction de fin d'instant.

Propriété 6.8 (Préservation du typage de $\longrightarrow_{\text{eoi}}$)**.** *Si* $\Sigma, s \vdash e/\mathcal{S} : ct \mid cf$ *et* $s, C \vdash e/\mathcal{S} \longrightarrow_{eoi} e'/\mathcal{S}'$ *avec* $C \subseteq \mathit{Clocks}(s)$*, alors il existe* Σ' *et* cf' *tels que* $\Sigma', s \vdash e'/\mathcal{S}' : ct \mid cf'$.

Démonstration. Par induction sur la dérivation de typage. On utilise le lemme 6.4 pour garantir que l'environnement de signaux $\mathcal{S}'$ est bien typé dans le cas de la règle ParentEoi. □

On peut maintenant montrer la même propriété dans le cas de la réduction d'instant :

Propriété 6.9 (Préservation du typage de $\xrightarrow{s}$)**.** *Si* $\Sigma, s \vdash e/\mathcal{S} : ct \mid cf$ *et* $e/\mathcal{S} \xrightarrow{s} e'/\mathcal{S}'$*, alors il existe* Σ' *et* cf' *tels que* $\Sigma', s \vdash e'/\mathcal{S}' : ct \mid cf'$

Démonstration. On fait la preuve par induction sur la taille de la dérivation de typage de e. Il suffit d'appliquer l'hypothèse d'induction dans la plupart des cas. Les cas non immédiats sont les suivants :

Cas $\texttt{domain}\ x\ \texttt{by}\ j\ \texttt{do}\ e_1$**.** L'idée principale de la preuve de ce cas est que puisque la variable d'horloge γ associée à l'horloge locale du domaine n'apparait ni dans l'environnement de typage Γ, ni dans le type ct du corps du domaine, on peut la substituer par un nom frais d'horloge ck' en gardant un terme du même type.

Plus formellement, on a :

$$\frac{i > 0 \qquad ck' \notin \mathit{Clocks}(\mathcal{S}) \qquad \mathcal{S}' = \mathcal{S} \sqcup \{s :: (ck', 0) \mapsto []\}}{\texttt{domain}\ x\ \texttt{by}\ i\ \texttt{do}\ e_1/\mathcal{S} \xrightarrow{s} e'/\mathcal{S}'}$$

avec $e' = e_1\ \texttt{in}\ (ck', 0/j)$. De la règle Domain, on déduit que :

$$\frac{\Gamma_0, \Sigma, \gamma \vdash e_1 : ct \mid cf_1 \qquad \Gamma_0, \Sigma, ce \vdash e_b : \texttt{int} \mid \emptyset \qquad \gamma \notin \mathit{ftv}(ct)}{\Gamma_0, \Sigma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1 : ct \mid cf}$$

avec $cf = cf_1 \setminus \{\gamma\}$. Notons $\theta = [\gamma \leftarrow ck']$. En appliquant le lemme 6.7, on obtient $\theta(\Gamma_0), \theta(\Sigma), \theta(\gamma) \vdash e : \theta(ct) \mid \theta(cf_1)$. On a $\theta(\Gamma_0) = \Gamma_0$ et $\theta(\Sigma) = \Sigma$ car $\mathcal{S}$ ne contient

pas de variables d'horloge et que $\Sigma \vdash \mathcal{S}$. On a également $\theta(ct) = ct$ puisque $\gamma \notin ftv(ct)$. On a donc :

$$\frac{\Gamma_0, \Sigma, ck' \vdash e_1 : ct \mid \theta(cf_1)}{\Gamma_0, \Sigma, ce \vdash e_1 \texttt{ in } (ck', 0/j) : ct \mid cf}$$

car $\theta(cf_1) \setminus \{ck'\} = cf$. On conclut donc que $\Sigma', s \vdash e'/\mathcal{S}' : ct \mid cf'$ en prenant $\Sigma' = \Sigma[ck' \mapsto []]$ et $cf' = cf$.

Cas v `in` $(ck', i/j)$**.** L'exécution du domaine est terminée et il se réduit en une expression $e' = v$ avec $\mathcal{S}' = \mathcal{S}$. L'expression e' a le même type puisque le type d'une valeur ne dépend pas de l'horloge locale. En effet, on déduit de la règle IN que $\Gamma_0, \Sigma, ck' \vdash v : ct \mid cf_1$ avec $cf = cf_1 \setminus \{ck'\}$. Le lemme 6.2 implique que $cf_1 = cf = \emptyset$. On peut ensuite appliquer le lemme 6.3 pour obtenir que $\Gamma_0, \Sigma, ck \vdash v : ct \mid \emptyset$. Les autres conditions sont immédiates.

Cas e_1 `in` $(ck', i/j)$ **(LOCALSTEP).** On effectue ici une réduction du corps e_1 du domaine dans son horloge locale. On applique donc l'hypothèse d'induction sur le corps du domaine et il suffit ensuite de vérifier que les conditions d'échappement de portée sont toujours valables. Plus précisément, on déduit de la règle IN que $\Gamma_0, \Sigma, ck' \vdash e_1 : ct \mid cf_1$ avec $cf = cf_1 \setminus \{ck'\}$. La condition $cf \subseteq Clocks(s)$ implique que $cf_1 \subseteq Clocks(s :: (ck', i))$, donc $\Sigma, s :: (ck', i) \vdash e_1/\mathcal{S} : ct \mid cf_1$. On peut appliquer l'hypothèse d'induction pour obtenir $\Gamma_0, \Sigma', ck' \vdash e_1' : ct \mid cf_1'$. On peut alors appliquer la règle IN pour obtenir le résultat voulu, avec $cf' = cf_1' \setminus \{ck'\}$ qui vérifie bien $cf' \subseteq Clocks(s)$ puisque $cf_1' \subseteq Clocks(s :: (ck', i))$.

Cas e_1 `in` $(ck', i/j)$ **(LOCALEOI).** Il suffit d'appliquer la propriété 6.8 qui montre que la réduction de fin d'instant conserve le typage et le lemme 6.4 pour s'assurer que le nouvel environnement de signaux est bien typé.

Cas `signal` x $(\texttt{h} : b, \texttt{ck} : ck', \texttt{d} : v_d, \texttt{g} : v_g, \texttt{rck} : v_{rck})$ `in` e_1**.** Alors $e' = e_1[x \leftarrow n^{ck'}]$ et $\mathcal{S}' = \mathcal{S} \sqcup \{s_{\mid ck'} \mapsto \{n \mapsto (v_d, v_g, v_d, \emptyset, b, ck'')\}$. On déduit de la règle de typage que :

$$\Gamma_0[x \mapsto (ct_1, ct_2)\ \texttt{event}\{ck'|\}], \Sigma, ck \vdash e_1 : ct \mid cf_1$$

où $cf = cf_1 \cup \{ck'\}$. Soit $\Sigma' = \Sigma[ck' \mapsto \{n \mapsto (ct_1, ct_2)\ \texttt{event}\{ck'|\}\}]$. En appliquant les lemmes 6.5 et 6.6, on obtient $\Gamma_0, \Sigma', ck \vdash e_1[x \mapsto n^{ck'}] : ct \mid cf_1$ et on prouve que $\Sigma' \vdash \mathcal{S}'$.

Cas `emit` $n^{ck'}$ v**.** On a $e' = ()$ et $\mathcal{S}' = \mathcal{S} + \{s_{\mid ck'} \mapsto \{n \mapsto \{v\}\}\}$. La règle de typage nous donne :

$$\frac{\Gamma_0, \Sigma, ck \vdash n^{ck'} : (ct_1, ct_2)\ \texttt{event}\{ck'|\} \mid \emptyset \qquad \Gamma_0, \Sigma, ck \vdash v : ct_1 \mid \emptyset}{\Gamma_0, \Sigma, ck \vdash \texttt{emit } n^{ck'}\ v : \texttt{unit} \mid \{ck'\}}$$

où $\Sigma(ck')(n) = (ct_1, ct_2)\ \texttt{event}\{ck'|\}$. On montre alors que $\Gamma_0, \Sigma, ck \vdash () : \texttt{unit} \mid \emptyset$ et que $\Sigma \vdash \mathcal{S}'$.

□

Nous allons maintenant exprimer les propriétés de progrès. Il faudra en particulier montrer pour toutes les opérations accédant à un signal que l'horloge du signal fait bien partie des horloges accessibles, c'est-à-dire des horloges apparaissant dans la pile. Il suffira de remarquer que cette horloge est incluse dans l'effet de l'expression, dont on sait par le typage d'une configuration qu'il est inclus dans l'ensemble des horloges accessibles.

Propriété 6.10 (Progrès des expressions instantanées)**.** *Si $\Sigma, s \vdash e/\mathcal{S} : ct \mid cf$ et que e est instantanée (c'est-à-dire $0 \vdash e$), alors soit :*

- *e est une valeur*
- *Il existe $e'/\mathcal{S}'$ telle que $e/\mathcal{S} \xrightarrow{s} e'/\mathcal{S}'$*

Démonstration. On fait la preuve par induction sur la dérivation de typage e. Le seul cas non trivial est celui de `emit` e_1 e_2. Les règles de bonne formation (figure 6.1) montrent que e_1 et e_2 sont des expressions instantanées, donc on peut appliquer l'hypothèse d'induction. Si e_1 ou e_2 n'est pas une valeur, alors on peut appliquer la règle CONTEXT de la sémantique opérationnelle (figure 5.2b).

Sinon, on a $e_1 = n^{ck'}$ et $e_2 = v$. Pour pouvoir réduire cette expression, il faut alors prouver que l'horloge du signal est bien accessible, c'est-à-dire qu'elle appartient à l'ensemble $Clocks(s)$ des horloges de la pile courante. C'est le cas car l'horloge ck' appartient à l'effet cf, qui est inclus dans $Clocks(s)$ par définition du typage d'une configuration. La pile $s_{\restriction ck'}$ est donc bien définie, ce qui implique que l'on peut réduire e en $e' = ()$ en utilisant la règle EMIT (voir figure 5.2a). On utilise le même principe pour toutes les opérations accédant à un signal pour cette démonstration et la suivante. □

Propriété 6.11 (Progrès de $\xrightarrow{s}$)**.** *If* $\Sigma, s \vdash e/\mathcal{S} : ct \mid cf$*, alors soit :*
- e *est une valeur*
- $e/\mathcal{S}$ *est une configuration de fin d'instant pour* $\texttt{top}(s)$
- *Il existe* $e'/\mathcal{S}'$ *telle que* $e/\mathcal{S} \xrightarrow{s} e'/\mathcal{S}'$

Démonstration. On fait à nouveau la preuve par induction :

Cas $\texttt{domain}\ x\ \texttt{by}\ j\ \texttt{do}\ e_1$**.** Alors $e' = e_1\ \texttt{in}\ (ck', 0/j)$.

Cas $e_1\ \texttt{in}\ (ck', i/j)$**.** On peut montrer que $\Sigma, s \vdash e_1/\mathcal{S} : ct \mid cf_1$ comme dans la preuve de la propriété 6.9. Par induction, soit il existe e_1' et $\mathcal{S}'$ tels que $e_1/\mathcal{S} \xrightarrow{s'} e_1'/\mathcal{S}'$, auquel cas on peut appliquer la règle LOCALSTEP, soit e_1 est une expression de fin d'instant pour ck' et alors :
- Soit $s', \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1$: on peut commencer un nouvel instant local avec la règle LOCALEOI, qui est bien une réduction d'instant pour la pile s.
- Soit $s', \{ck'\}, \mathcal{S} \not\vdash_{\text{next}} e_1$: alors $e_1\ \texttt{in}\ (ck', i/j)/\mathcal{S}$ est une configuration de fin d'instant pour ck (la règle PARENTEOI s'applique forcément).

□

On peut maintenant exprimer la sûreté du calcul d'horloges proprement dite. On montre d'abord la sûreté de la réduction pour une pile quelconque :

Propriété 6.12 (Sureté de $\xrightarrow{s}$)**.** *Si* $\Sigma, s \vdash e/\mathcal{S} : ct \mid cf$*, alors soit :*
- *Il existe* $e'/\mathcal{S}'$ *telle que* $e/\mathcal{S} \xrightarrow{s}{}^* e'/\mathcal{S}'$ *et* $e'/\mathcal{S}'$ *est une configuration de fin d'instant pour* $\texttt{top}(s)$.
- *Pour chaque* $e'/\mathcal{S}'$ *telle que* $e/\mathcal{S} \xrightarrow{s}{}^* e'/\mathcal{S}'$*, il existe* $e''/\mathcal{S}''$ *telle que* $e'/\mathcal{S}' \xrightarrow{s} e''/\mathcal{S}''$.

On en déduit la sûreté de la réduction d'un programme dans l'horloge globale, qui constitue le théorème de sûreté du calcul d'horloges :

Théorème 6.13 (Sureté du calcul d'horloges)**.** *Si* $[], (\top_{ck}, 0) \vdash p/\mathcal{S}_0 : ct \mid cf$*, alors soit :*
- *Il existe* $v/\mathcal{S}$ *telle que* $p/\mathcal{S}_0 \Rightarrow^* v/\mathcal{S}$.
- *Pour chaque* $p'/\mathcal{S}'$ *telle que* $p/\mathcal{S}_0 \Rightarrow^* p'/\mathcal{S}'$*, il existe* $p''/\mathcal{S}''$ *telle que* $p'/\mathcal{S}' \Rightarrow p''/\mathcal{S}''$.

Remarque 8. On remarquera qu'il y a une condition que notre preuve de sûreté, et plus particulièrement la propriété 6.11 de progrès, ne garantit pas. Il s'agit du fait que la borne e_b dans $\texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1$ doit être strictement positive. Il peut donc se passer une erreur à l'exécution si ce n'est pas le cas. La sémantique intuitive que l'on pourrait donner au cas où $e_b \leq 0$ est que l'on n'exécute jamais d'instant local et que le domaine reste bloqué dans son état initial. Nous avons choisi de ne pas inclure ce cas dans la sémantique car il complexifie les règles pour un intérêt faible.

6.5 Limites du système de types

Le système de types que l'on vient de présenter garantit l'absence d'erreurs à l'exécution, c'est-à-dire qu'un signal n'échappe jamais de son domaine et que l'on ne dépend jamais instantanément d'un signal lent, mais rejette trop de programmes. Le premier problème se pose lorsque l'on veut écrire des combinateurs utilisant des domaines réactifs :

exemples/ch6/run_domain.rml

```
let process run_domain q =
  domain ck do run q done
The clock of this expression, that is, '_a process {?ck|''_e0}
depends on ck which escapes its scope.
```

Ce programme est rejeté car l'horloge d'activation du processus `q` est égale à l'horloge locale à l'endroit où on le lance, c'est-à-dire `?ck`. `?ck` échappe donc de son domaine. L'origine de ce problème est qu'en ML, le type des arguments d'une fonction est un type ct, pas un schéma de type cs, comme on peut le vérifier sur la figure 6.2. Si l'argument d'une fonction est un processus, alors son horloge d'activation ne peut pas être quantifiée universellement, comme c'est le cas pour un processus déclaré dans un `let`. Pour résoudre ce problème, il faut rendre possible l'utilisation d'un schéma de type pour les arguments des fonctions, ce que l'on appelle du *polymorphisme de rang supérieur* [Rey74, Gir72]. Nous reviendrons sur les solutions possibles pour ce problème dans la partie 8.1.

Le second problème du calcul d'horloges se pose lorsque l'on utilise des fonctions ou des processus de première classe, par exemple pour les stocker dans une liste :

```
signal s default 0 gather (+)
val s : (int, int list) event{?topck|}

let pure x = x + 1
val pure :: int =>{0} int

let unpure x = emit s x; x + 1
val unpure :: int =>{?topck} int

let l = [pure; unpure]
This expression has clock ((int =>{?topck} int) list),
but is used with clock ((int =>{0} int) list).
```

On voit que l'on est incapable de stocker une fonction sans effet avec une fonction qui a un effet puisqu'il est impossible d'unifier leurs effets, qui doivent être égaux. Cela pose un problème même pour des programmes n'utilisant pas de domaines réactifs, pour lesquels le calcul d'horloges n'apporte rien de plus que le système de types habituel. Par exemple, on est incapable de donner un type au processus `proxy` (ligne 15 de la figure 2.3, page 23) de notre mini-bibliothèque d'agents puisqu'il utilise au même endroit une fonction avec effet et une fonction sans effet. Pour lever cette restriction, on peut utiliser une forme de sous-typage restreint aux effets qui se nomme *subeffecting* [NNH99]. Nous montrerons comment procéder dans la partie 8.3.

CHAPITRE 7

Analyse de réactivité

Nous présentons dans ce chapitre une analyse de réactivité. Son but est de garantir que le programme est *réactif*, ou *coopératif*, c'est-à-dire que chaque instant termine. Elle permet donc d'éviter qu'un processus ou un domaine réactif empêche les autres processus de s'exécuter en faisant une boucle instantanée infinie. Cette analyse se base sur l'approche de [ANN99], qui consiste à utiliser un langage d'effets qui abstrait la structure du programme, que l'on appelle des *comportements*. L'utilisation d'un système de types permet d'obtenir une analyse compatible avec l'ordre supérieur, simple à comprendre et à implémenter et efficace. Nous montrerons ensuite la sûreté de l'analyse, c'est-à-dire que tout programme bien typé est réactif. Une version de cette analyse réduite à REACTIVEML classique a été présentée dans [MP13a] et dans [MP13b] avec la preuve de sûreté correspondante. Nous conseillons la lecture de ces articles au lecteur qui n'est pas intéressé par la gestion des domaines réactifs, puisque celle-ci complexifie notoirement la présentation. Comme dans le cas du calcul d'horloges, nous présentons d'abord une version simple que nous étendrons dans le chapitre 8.

7.1 Intuitions et limites

Réactivité en REACTIVEML classique Le moteur d'exécution de REACTIVEML se base sur un *ordonnancement coopératif*, c'est-à-dire qu'il repose sur le fait que chaque processus doit rendre la main à l'ordonnanceur à chaque instant pour laisser les autres processus s'exécuter. Ce type d'ordonnancement prend son origine dans la notion de *coroutines* [Con63], c'est-à-dire de fonctions avec plusieurs points d'entrée et de sortie, et de *trampoline* [GFW99], c'est-à-dire de fonctions renvoyant des continuations avec la suite du calcul. L'ordonnancement coopératif permet d'implémenter la concurrence efficacement de façon séquentielle, sans les problèmes liés à l'exécution parallèle comme la gestion des accès concurrents aux ressources partagées. Il est utilisé par de nombreuses autres bibliothèques de threads légers, comme les *calculs asynchrones* [SPL11] en F#, CONCURRENT HASKELL [JGF96] ou encore les bibliothèques LWT [Vou08] et ASYNC [1] en OCAML. La contrepartie de cette méthode d'implémentation est que c'est au programmeur de s'assurer que les processus mis en parallèle vont bel et bien coopérer. Dans les bibliothèques de programmation coopérative, cela se traduit par des bonnes pratiques de programmation de la responsabilité du programmeur. Ainsi, on peut lire par exemple dans la documentation de LWT : [2]

> Rules : `Lwt` will always try to execute as much as possible before yielding and switching to another cooperative thread. In order to make it work well, you must follow the following rules :
> – do not write function that may take time to complete without using `Lwt`,
> – do not do IOs that may block, otherwise the whole program will hang. You must instead use asynchronous IOs operations.

1. https://bitbucket.org/yminsky/ocaml-core/wiki/DummiesGuideToAsync
2. http://ocsigen.org/lwt/manual/

L'analyse statique que nous proposons dans ce chapitre permet de garantir qu'un programme est réactif au moment de la compilation. Le modèle de concurrence synchrone permet en outre de donner une condition simple pour qu'un programme soit réactif : il suffit que tous les processus coopèrent à chaque instant.

Il y a deux sources de non-réactivité en REACTIVEML : les boucles instantanées et les récursions instantanées. On a déjà vu un exemple de boucle instantanée dans le chapitre 3 avec le processus `timer` (page 38) :

exemples/ch3/timer.rml

```
let process timer delay s =
  let time = ref (Unix.gettimeofday ()) in
  loop
    let time' = Unix.gettimeofday () in
    if time' -. !time >= delay
    then (emit s (); time := time')
  end
```

```
Warning: This expression may be an instantaneous loop.
```

Le corps de la boucle termine instantanément. On recommence donc une nouvelle instance de la boucle qui termine aussi instantanément et ainsi de suite. L'analyse détecte cette boucle instantanée et affiche un message d'avertissement. On peut obtenir le même comportement avec un processus récursif, comme le processus `instantaneous` suivant, que notre analyse sait aussi gérer :

exemples/ch3/instantaneous.rml

```
let rec process instantaneous s =
  emit s (); run (instantaneous s)
```

```
Warning: This expression may produce an instantaneous recursion.
```

Approche et limites L'analyse proposée ici se fonde sur le constat suivant : il suffit qu'il y ait toujours au moins un instant entre l'instanciation d'un processus et son appel récursif pour garantir que ce processus est réactif. Dans le cas des boucles inconditionnelles, il faut que l'exécution du corps prenne au moins un instant. Notre analyse vérifie ces conditions statiquement.

Il s'agit de conditions très fortes qui sont loin d'être nécessaires pour la réactivité d'un processus. Par exemple, notre analyse va refuser l'exécution d'un `map` en parallèle :

exemples/ch7/par_map.rml

```
let rec process par_map p l = match l with
| [] -> []
| x :: l -> let x' = run p x
            and l' = run par_map p l in
            x' :: l'
```

```
Warning: This expression may produce an instantaneous recursion.
```

Ce processus fait des appels récursifs instantanés, mais il est réactif car la récursion termine si la liste est finie. Comme le langage permet de créer des structures de données mutables et récursives, il peut être difficile de prouver la terminaison de tels processus. Par exemple, le processus suivant ne termine jamais :

```
let rec l = 0 :: l in run par_map p l
```

Par conséquent, l'analyse statique ne rejette pas de programmes et affiche simplement des messages d'avertissement. Puisque l'on considère toutes les fonctions ML comme instantanées, elles sont réactives si et seulement si elles terminent. Nous ne cherchons pas à garantir la terminaison des fonctions ML, ce qui nécessiterait une analyse plus fine ou de réduire l'expressivité du langage à la manière de COQ [BFG+04]. Nous allons donc restreindre notre analyse aux processus et faire l'hypothèse que toutes les fonctions terminent. Cela permet d'éviter l'affichage d'un message d'avertissement pour chaque définition de fonction récursive. Cette hypothèse est rendue possible par la distinction syntaxique entre fonctions et processus que garantit l'analyse de bonne formation des expressions (partie 6.2).

On choisit également de ne pas traiter le cas des fonctions bloquantes, comme par exemple les fonctions d'entrées/sorties, qui peuvent également rendre un programme non réactif. En effet, ces fonctions ne doivent jamais être utilisées dans le cadre de l'ordonnancement coopératif. Il faudrait les remplacer par des entrées/sorties coopératives en suivant ce qui est fait dans [MJT04].

Notre analyse ne prend pas non plus en compte la présence des signaux. Elle sur-approxime les différents comportements possibles, comme le montre l'exemple suivant :

exemples/ch7/imprecise.rml

```
let rec process imprecise =
  signal s in
  present s then () else ();
  run imprecise
```

```
Warning: This expression may produce an instantaneous recursion.
```

Comme le signal local `s` n'est jamais émis, seule la branche **`else`** est exécutée à l'instant suivant (rappelons que, en REACTIVEML, on doit attendre la fin de l'instant pour décider de l'absence d'un signal). Ce processus est donc coopératif, mais l'analyse va le rejeter car elle considère que le signal pourrait être présent, auquel cas le processus `imprecise` s'appellerait récursivement de façon instantanée.

Enfin, ce n'est pas parce que nous garantissons que le programme est réactif qu'il est temps-réel, c'est-à-dire qu'il réagit en temps et mémoire bornés. On peut l'observer par exemple sur le processus `server` (page 19) :

```
let rec process server add =
  await add(p, ack) in
  run server add
  ||
  let v = run p in emit ack v
```

Comme `server` exécute un processus en parallèle à chaque fois que le signal `add` est émis, ce programme peut exécuter un nombre arbitraire de processus. Il n'est donc pas temps-réel, mais est réactif puisque l'attente de la valeur d'un signal prend un instant.

Domaines réactifs non coopératifs Nous avons vu dans le chapitre 3 qu'un domaine réactif non borné pouvait également rendre un programme non réactif, comme dans le cas du processus `nonreactive_domain` :

exemples/ch3/nonreactive_domain.rml

```
let process nonreactive_domain =
  domain ck do
    loop pause ck end
  done
```

```
Warning: This reactive domain may not cooperate.
```

Le domaine réactif d'horloge `ck` n'attend jamais son horloge parente, donc il se comporte comme une boucle infinie instantanée. On dit aussi que le domaine est non coopératif. On peut corriger ce problème en rendant le domaine périodique :

exemples/ch7/periodic_domain.rml

```
let process periodic_domain =
  domain ck by 2 do
    loop pause ck end
  done
```

Le domaine coopère maintenant tous les deux instants locaux et ne pose plus de problème. Il faut quand même noter que ce domaine réactif est désormais équivalent à une boucle infinie sur son horloge parente. En effet, il exécute deux instants locaux puis attend le prochain instant de son horloge parente. Il n'attend jamais une horloge plus lente. Cela peut à nouveau poser problème si on lance le processus `periodic_domain` à l'intérieur d'un autre domaine non borné, qui devient alors non coopératif :

exemple/ch7/nested_domains.rml
```
let process nested_domain =
  domain ck2 do
    run periodic_domain
  done
```

```
Warning: This reactive domain may not cooperate.
```

Le domaine réactif d'horloge ck2 n'est pas coopératif car le processus periodic_domain n'attend jamais une horloge plus lente. Notre analyse peut également détecter ce cas de figure. Une autre façon de corriger le processus nonreactive_domain est d'utiliser l'opérateur **quiet pause** :

exemples/ch7/qpause_domain.rml
```
let process qpause_domain =
  domain ck do
    loop quiet pause ck end
  done
```

Une boucle infinie qui utilise l'opérateur **quiet pause** ne rend pas le domaine non coopératif puisqu'elle n'influence pas le choix d'exécuter ou non un autre instant local. C'est pourquoi l'analyse n'affiche pas d'avertissement.

Notre analyse doit également prendre en compte le fait qu'un domaine peut terminer instantanément, comme dans l'exemple suivant :

exemples/ch7/instantaneous_domain.rml
```
let process instantaneous_domain =
  loop
    domain ck do
      pause ck; pause ck
    done
  end
```

```
Warning: This expression may be an instantaneous loop.
```

Ce domaine réactif exécute trois instants locaux avant de terminer. Il termine instantanément du point de vue de son domaine parent, ce qui fait que la boucle est instantanée. Ce n'est plus le cas si l'on force ce domaine à avoir une période égale à 2 :

exemples/ch7/instantaneous_domain_period.rml
```
let process instantaneous_domain_period =
  loop
    domain ck by 2 do
      pause ck; pause ck
    done
  end
```

```
Warning: This expression may be an instantaneous loop.
```

Puisque le domaine réactif n'exécute que deux instants locaux par instant de son horloge parente, son exécution prend un instant de son horloge parente, ce qui rend la boucle non instantanée. Notre analyse ne peut pas détecter ce cas de figure car la réactivité du domaine dépend de la valeur de la période. En effet, la boucle serait instantanée si on remplaçait la période par n'importe quelle valeur plus grande que 2. On choisit donc de sur-approximer les comportements possibles et d'afficher un avertissement dans le cas du processus instantaneous_domain_period, bien qu'il soit réactif.

En résumé, pour gérer le cas des domaines réactifs, notre analyse doit donc :

- Garantir que si un domaine réactif n'est pas borné, alors on ne boucle jamais sur son horloge locale. On vérifie pour cela qu'il n'exécute qu'un nombre fini d'instants locaux avant d'attendre une horloge plus lente.
- Traduire le comportement des domaines réactifs vis-à-vis de leur horloge parente. Il faut donc que l'analyse puisse détecter si un domaine réactif peut potentiellement terminer instantanément ou à l'opposé boucler infiniment sur son horloge parente. Il faut en particulier prendre en compte le fait que si un domaine réactif est périodique, alors une boucle sur son horloge locale correspond à une boucle sur son horloge parente.

7.2 Le langage des comportements

Nous allons maintenant définir un langage pour exprimer le comportement réactif des processus. On appelle *comportements* les termes de ce langage, que l'on note κ. L'idée est d'oublier complètement les valeurs mais de garder une abstraction de la structure du programme.

Le langage

Le langage des comportements est défini par :

$$\kappa ::= 0 \mid ce \mid \phi \mid \kappa \mid\mid \kappa \mid \kappa + \kappa \mid \kappa;\kappa \mid \mu\phi.\,\kappa \mid \texttt{run}\ \kappa \mid \texttt{domain}\ \kappa \qquad \text{(comportements)}$$

$$ce ::= \gamma \mid ck \qquad \text{(horloges)}$$

L'ordre de priorité des opérateurs est le suivant (du plus au moins prioritaire) : `run`, `domain`, ;, $+$, $\mid\mid$ et enfin μ. Par exemple : $\mu\phi.\,\kappa_1 \mid\mid \texttt{run}\ \kappa_2 + ck;\kappa_3$ signifie $\mu\phi.\,(\kappa_1 \mid\mid ((\texttt{run}\ \kappa_2) + (ck;\kappa_3)))$.

On note 0 les actions potentiellement instantanées, comme l'appel d'une fonction ML ou l'émission d'un signal. On note ce une action surement non instantanée qui prend au moins un instant de l'horloge ce, comme la construction `pause` ce. On utilise également des variables ϕ pour représenter le comportement de processus pris en argument, puisque les processus sont des objets de première classe.

Afin d'obtenir une analyse correcte et précise, les comportements doivent traduire la structure du programme, en commençant par la composition parallèle synchrone. On peut le voir sur l'exemple du combinateur `par_comb` qui prend en argument deux processus et les lance en parallèle dans une boucle :

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
```

La composition parallèle termine lorsque les deux branches ont terminé. Cela signifie que la boucle est instantanée si `q1` ou `q2` est non instantané. Pour prendre en compte ce comportement, on ajoute dans le langage des comportements la composition parallèle de comportements que l'on note $\kappa_1 \mid\mid \kappa_2$. Le comportement associé à `par_comb` est donc `run` $\phi_1 \mid\mid$ `run` ϕ_2 où ϕ_1 (resp. ϕ_2) est le comportement de `q1` (resp. `q2`). Le comportement `run` est associé au lancement d'un processus et nous justifierons sa présence au moment de la définition du système de types. De la même façon, on peut définir un autre combinateur qui lance un de ses deux arguments selon la valeur d'une condition `c` :

```
let process if_comb c q1 q2 =
  loop
    if c then run q1 else run q2
  end
```

Dans le cas du processus `if_comb`, il faut que les deux processus `q1` et `q2` soient non instantanés pour que la boucle soit non instantanée. On utilise un opérateur de choix non déterministe, noté $\kappa_1 + \kappa_2$, pour représenter les différentes alternatives dans l'exécution de ce processus. On voit bien comment on abstrait les valeurs : on ne garde que les différentes alternatives et on oublie les conditions. On doit également ajouter une notion de séquence dans les comportements, que l'on note $\kappa_1;\kappa_2$, puisque l'on doit conserver l'ordre entre les actions, comme le montre l'exemple suivant :

```
let rec process good_rec = pause global_ck; run good_rec

let rec process bad_rec = run bad_rec; pause global_ck
```

L'ordre entre l'appel de `pause` et l'appel récursif est capital puisque le processus `good_rec` est bien réactif, alors que `bad_rec` boucle instantanément. En outre, puisque `good_rec` est défini récursivement, son comportement κ doit vérifier $\kappa = \top_{ck};\texttt{run}\ \kappa$. On résout cette équation en introduisant

un opérateur de récursion explicite μ dans les comportements. On a ainsi $\kappa = \mu\phi.\, \top_{ck}; \texttt{run}\ \phi$. Les comportements récursifs vérifient les propriétés habituelles :

$$\mu\phi.\, \kappa = \kappa[\phi \leftarrow \mu\phi.\, \kappa] \qquad\qquad \mu\phi.\, \kappa = \kappa \text{ si } \phi \notin \mathit{fbv}(\kappa)$$

où $\mathit{fbv}(\kappa)$ est l'ensemble des variables de comportement libres dans κ. On peut remarquer qu'il n'y a pas d'opérateur pour représenter le comportement d'une boucle. En effet, une boucle `loop` e est un cas particulier de processus récursif :

$$\texttt{loop}\ e \triangleq \texttt{run}\ ((\texttt{rec}\ \mathit{loop} = \lambda x.\texttt{process}\ (\texttt{run}\ x; \texttt{run}\ (\mathit{loop}\ x)))\ (\texttt{process}\ e))$$

Le comportement d'une boucle, noté κ^∞, est donc également un cas particulier de comportement récursif défini par :

$$\kappa^\infty \triangleq \mu\phi.\, \kappa; \texttt{run}\ \phi$$

Enfin, la dernière construction de notre langage de comportements est l'opérateur `domain`, qui est le comportement associé à un domaine réactif. La justification de cet opérateur sera donnée au moment de la preuve de sûreté de l'analyse dans la partie 7.4.

Comportements réactifs

Maintenant que nous avons défini le langage des comportements, nous allons pouvoir caractériser les comportements que nous souhaitons rejeter, c'est-à-dire les boucles et les récursions instantanées. Nous parlerons du cas des domaines non coopératifs dans la partie suivante. On vérifie en fait une condition plus forte, suffisante et plus simple à vérifier : il doit se passer au moins un instant entre l'instanciation d'un processus et un appel récursif. Nous verrons dans la suite que cette condition permet de traiter aussi bien le cas des récursions instantanées, des boucles instantanées que des domaines réactifs non coopératifs. L'exemple du processus par_map (page 110) a montré que cette condition n'est bien sûr pas nécessaire pour qu'un processus soit coopératif.

Avant de pouvoir définir formellement les comportements que l'on dira réactifs, nous devons d'abord définir la notion de comportement *non instantané*, associé à un processus dont l'exécution prend au moins un instant :

Définition 9 (Comportement non instantané)**.** Un comportement est *non instantané* dans l'ensemble d'horloges C, ce que l'on note $\texttt{noinst}_C(\kappa)$, si toutes les alternatives prennent au moins un instant d'une horloge dans C, c'est-à-dire si elles contiennent au moins un comportement non instantané ck avec $ck \in C$:

$$\frac{ck \in C}{\texttt{noinst}_C(ck)} \qquad \frac{}{\texttt{noinst}_C(\gamma)} \qquad \frac{}{\texttt{noinst}_C(\phi)} \qquad \frac{\texttt{noinst}_C(\kappa_1)}{\texttt{noinst}_C(\kappa_1 \mid\mid \kappa_2)} \qquad \frac{\texttt{noinst}_C(\kappa_2)}{\texttt{noinst}_C(\kappa_1 \mid\mid \kappa_2)}$$

$$\frac{\texttt{noinst}_C(\kappa_1) \qquad \texttt{noinst}_C(\kappa_2)}{\texttt{noinst}_C(\kappa_1 + \kappa_2)} \qquad \frac{\texttt{noinst}_C(\kappa_1)}{\texttt{noinst}_C(\kappa_1; \kappa_2)} \qquad \frac{\texttt{noinst}_C(\kappa_2)}{\texttt{noinst}_C(\kappa_1; \kappa_2)} \qquad \frac{\texttt{noinst}_C(\kappa)}{\texttt{noinst}_C(\mu\phi.\, \kappa)}$$

$$\frac{\texttt{noinst}_C(\kappa)}{\texttt{noinst}_C(\texttt{run}\ \kappa)} \qquad\qquad \frac{\texttt{noinst}_C(\kappa)}{\texttt{noinst}_C(\texttt{domain}\ \kappa)}$$

On considère comme non instantanées les horloges ck de l'ensemble C, les variables d'horloges γ et les variables de comportement ϕ. Dans le cas de la composition parallèle $\kappa_1 \mid\mid \kappa_2$ et de la séquence $\kappa_1; \kappa_2$, il suffit qu'un des deux comportements soit non instantané, alors que pour le choix $\kappa_1 + \kappa_2$ il faut que les deux soient non instantanés. Un comportement récursif $\mu\phi.\, \kappa$ est non instantané si son corps κ est non instantané, et de même pour les comportements `run` et `domain`.

L'ensemble C correspond dans le cas général aux horloges contenues dans une pile. Le prédicat est alors vrai pour les comportements qui prennent au moins un instant de l'horloge au sommet de la pile. On ne prend pas en compte les horloges plus rapides qui n'apparaissent pas dans cet

ensemble C et que l'on considère comme instantanées. Le fait que les variables de comportement soient non instantanées signifie que l'on considère par défaut les processus pris en paramètre comme non instantanés. Si jamais ce n'est pas le cas, alors la vérification de réactivité se fera une fois la variable instanciée avec le comportement du processus.

Par exemple, le comportement $ck;0$ associé à l'expression `pause` $ck;0$ est non instantané dans l'ensemble $C = \{ck\}$, ce que l'on note $\texttt{noinst}_{\{ck\}}(ck;0)$. A l'opposé, le comportement $0 + (ck;0)$ associé à l'expression `if` $x > 0$ `then` $x - 1$ `else` (`pause` $ck;0$) est potentiellement instantané quel que soit l'ensemble C choisi.

On dit qu'un comportement est *réactif* si pour chaque récursion $\mu\phi.\,\kappa'$ qu'il contient, la variable de récursion ϕ n'apparait pas dans le premier instant du corps κ' :

Définition 10 (Comportement réactif)**.** Un comportement κ est *réactif* dans l'ensemble d'horloges C s'il vérifie $C, \emptyset \vdash \kappa$, où le prédicat $C, R \vdash \kappa$ est défini pour un ensemble d'horloges C et un ensemble de variables de comportements R par :

$$\frac{}{C, R \vdash 0} \qquad \frac{}{C, R \vdash ce} \qquad \frac{\phi \notin R}{C, R \vdash \phi} \qquad \frac{C, R \vdash \kappa_1 \qquad C, R \vdash \kappa_2}{C, R \vdash \kappa_1 \mid\mid \kappa_2} \qquad \frac{C, R \vdash \kappa_1 \qquad C, R \vdash \kappa_2}{C, R \vdash \kappa_1 + \kappa_2}$$

$$\frac{C, R \vdash \kappa_1 \qquad \texttt{noinst}_C(\kappa_1) \qquad C, \emptyset \vdash \kappa_2}{C, R \vdash \kappa_1; \kappa_2} \qquad \frac{C, R \vdash \kappa_1 \qquad \mathit{not}(\texttt{noinst}_C(\kappa_1)) \qquad C, R \vdash \kappa_2}{C, R \vdash \kappa_1; \kappa_2}$$

$$\frac{C, R \cup \{\phi\} \vdash \kappa}{C, R \vdash \mu\phi.\,\kappa} \qquad \frac{C, R \vdash \kappa}{C, R \vdash \texttt{run}\ \kappa} \qquad \frac{C, R \vdash \kappa}{C, R \vdash \texttt{domain}\ \kappa}$$

Le prédicat $C, R \vdash \kappa$ signifie que le comportement κ est réactif vis-à-vis des variables de comportement dans l'ensemble R, c'est-à-dire que ces variables n'apparaissent pas dans son premier instant et que toutes les récursions contenues dans κ sont non instantanées. Ainsi, un comportement instantané 0 ou associé à une horloge ce est toujours réactif. Une variable de comportement ϕ est réactive si elle n'appartient pas à l'ensemble R. On ajoute la variable ϕ d'un comportement récursif $\mu\phi.\,\kappa$ à l'ensemble R au moment de vérifier la réactivité du corps κ pour indiquer que cette variable ne doit pas apparaître dans le premier instant du corps. Pour la composition parallèle $\kappa_1 \mid\mid \kappa_2$ et le choix $\kappa_1 + \kappa_2$, il faut que les deux comportements soient réactifs. De même, le corps d'un comportement `run` κ ou `domain` κ doit être réactif.

Le cas le plus intéressant est celui de la séquence. Si le comportement κ_1 de la première partie est non instantané et vérifie $C, R \vdash \kappa_1$, alors on sait que les variables contenues dans R n'apparaissent pas dans le premier instant de la séquence. Il ne faut quand même pas oublier de vérifier les récurrences de κ_2, c'est-à-dire que κ_2 est réactif.

On peut vérifier que cette définition d'un comportement réactif traite bien le cas des boucles. En effet, le comportement d'une boucle $\kappa^\infty = \mu\phi.\,\kappa;\texttt{run}\ \phi$ est réactif si et seulement si κ est non instantané au sens de la définition 9.

Comportement des domaines réactifs

Après avoir vu comment nous allions traiter le cas des récursions et boucles instantanées, nous allons maintenant nous intéresser au comportement des domaines réactifs. Pour représenter leur comportement, nous allons utiliser l'opérateur `domain` mais aussi modifier le comportement du corps du domaine pour que l'horloge locale γ n'apparaisse pas dans le comportement du domaine. Dans le cas d'un domaine non périodique, on obtient le comportement du domaine en remplaçant γ par le comportement instantané 0 dans le comportement du corps du domaine. Cela revient à masquer les instants locaux. Dans l'exemple du processus `nonreactive_domain`, le corps du domaine a un comportement γ^∞, donc le domaine a le comportement `domain` (0^∞) qui n'est pas réactif. On détecte ainsi le fait que le domaine n'est pas coopératif. Si on considère maintenant le processus `instantaneous_domain`, le corps a le comportement $\gamma;\gamma$, donc le domaine

a le comportement $\texttt{domain}\ (0;0)$. Ce comportement est instantané, ce qui permet de détecter la boucle instantanée.

On traite aussi simplement l'opérateur qpause. Alors que l'appel de pause ck' a le comportement ck', l'appel de qpause ck' a toujours le comportement $\top_{ck}$. Ainsi, le processus qpause_domain a le comportement $\texttt{domain}\ (\top_{ck}^{\infty})$ qui est bien réactif.

Le cas des domaines réactifs périodiques est plus complexe. Il faut en effet traduire dans le comportement du domaine le fait qu'une boucle sur l'horloge locale correspond à une boucle sur l'horloge parente. On définit pour cela une opération de substitution conditionnelle un peu étrange au premier abord, qui nous servira à exprimer le comportement d'un domaine réactif périodique d'horloge ck'. L'idée est de remplacer toutes les occurrences de ck' par le comportement instantané 0, mais en ajoutant la propriété que ck'^{∞} est équivalent à ck^{∞}, c'est-à-dire qu'il ne peut y avoir qu'un nombre fini d'instants de ck' par instant de ck.

Définition 11. L'opération $\kappa[ck' \leftarrow 0 \mid ck]$ est définie uniquement pour les comportements κ réactifs (tels que $Clocks(\kappa), \emptyset \vdash \kappa$) par :

$$\kappa[ck' \leftarrow 0 \mid ck] \triangleq \kappa \quad \text{si } ck' \notin Clocks(\kappa)$$

$$\big(G[ck']\big)\,[ck' \leftarrow 0 \mid ck] \triangleq \begin{cases} G[0][ck' \leftarrow 0 \mid ck] & \text{si } Clocks(\kappa), \emptyset \vdash G[0] \\ G[ck][ck' \leftarrow 0 \mid ck] & \text{sinon} \end{cases}$$

où les contextes G sont définis par :

$$G ::= [] \mid G \,||\, \kappa \mid \kappa \,||\, G \mid G + \kappa \mid \kappa + G \mid G;\kappa \mid \kappa;G \mid \texttt{run}\ G \mid \texttt{domain}\ G$$

On substitue toutes les occurrences de ck' par 0, sauf si cela rend le comportement non réactif. Dans ce cas, cela signifie que le comportement contient une boucle infinie sur ck'. On remplace alors ck' par ck pour la transformer en une boucle infinie sur ck. On a par exemple :

$$\big(ck'^{\infty}\big)\,[ck' \leftarrow 0 \mid ck] = ck^{\infty}$$
$$\big(ck';ck'\big)\,[ck' \leftarrow 0 \mid ck] = 0;0$$

Le premier exemple correspond au cas du processus periodic_domain. Le corps du domaine d'horloge ck' boucle sur l'horloge locale, donc le domaine boucle sur son horloge parente. Le second exemple correspond au cas du processus instantaneous_domain_period. Le corps du domaine fait trois instants locaux, donc le domaine a un comportement instantané. On sur-approxime le comportement du domaine, qui est en fait coopératif, puisque l'on ne regarde pas la valeur de la période.

Équivalence de comportements

On définit également une relation d'équivalence $\equiv$ sur les comportements. Elle permet intuitivement de simplifier les comportements, en montrant par exemple que les comportements $0;(0+0)$ et 0 sont équivalents. Il s'agit de la plus petite relation d'équivalence vérifiant :

- $\equiv$ est une relation d'équivalence, c'est-à-dire une relation réflexive, transitive et symétrique.
- Les opérateurs $||$, $+$ et $;$ sont compatibles avec cette relation, idempotents et associatifs. $||$ et $+$ sont commutatifs, mais pas $;$.
- Le comportement 0 est l'élément neutre de $||$.
- La relation $\equiv$ vérifie également :

$$\frac{\kappa_1 \equiv \kappa_2}{\mu\phi.\,\kappa_1 \equiv \mu\phi.\,\kappa_2} \qquad \frac{\kappa_1 \equiv \kappa_2}{\texttt{run}\ \kappa_1 \equiv \texttt{run}\ \kappa_2} \qquad \frac{\kappa_1 \equiv \kappa_2}{\texttt{domain}\ \kappa_1 \equiv \texttt{domain}\ \kappa_2}$$

A l'aide de toutes ces propriétés, on peut par exemple montrer que :

$$\mu\phi.\,((ce \,||\, 0);(\texttt{run}\ \phi + \texttt{run}\ \phi)) \equiv \mu\phi.\,ce;\texttt{run}\ \phi$$

Une propriété importante de cette relation est qu'elle préserve la réactivité :

Propriété 7.1. *Si $\kappa_1 \equiv \kappa_2$ et $C, R \vdash \kappa_1$, alors $C, R \vdash \kappa_2$.*

Notations

Nous avons besoin de plusieurs autres notations pour la preuve de sûreté de l'analyse. Cela commence par la taille d'un comportement :

Définition 12 (Taille d'un comportement)**.** La taille d'un comportement κ, que l'on note $|\kappa|$, est définie par :

$$
\begin{aligned}
|0| = |ce| = |\phi| &= 1 \\
|\kappa_1 \mid\mid \kappa_2| = |\kappa_1 + \kappa_2| = |\kappa_1; \kappa_2| &= 1 + |\kappa_1| + |\kappa_2| \\
|\texttt{run}\ \kappa| = |\texttt{domain}\ \kappa| &= 1 + |\kappa| \\
|\mu\phi.\, \kappa| &= \begin{cases} \infty & \text{si } \phi \in \mathit{fbv}(\kappa) \\ |\kappa| & \text{sinon} \end{cases}
\end{aligned}
$$

Un comportement κ est *fini* si $|\kappa| < \infty$. Par exemple, $|ck; 0| = 3$ et $|\mu\phi.\, (0; \phi)| = \infty$.

On définit également un opérateur qui extrait le premier instant d'un comportement :

Définition 13 (Premier instant d'un comportement)**.** Le premier instant d'un comportement dans la pile s, noté $\texttt{fst}_s(\kappa)$, est la partie du comportement décrivant le premier instant de l'horloge $\texttt{top}(s)$. Plus formellement, cette opération est définie par :

$$
\begin{aligned}
\texttt{fst}_s(0) &= 0 \\
\texttt{fst}_s(ck) &= \begin{cases} 0 & \text{si } ck \in \mathit{Clocks}(s) \\ ck & \text{sinon} \end{cases} \\
\texttt{fst}_s(\gamma) &= 0 \\
\texttt{fst}_s(\phi) &= \phi \\
\texttt{fst}_s(\kappa_1 \mid\mid \kappa_2) &= \texttt{fst}_s(\kappa_1) \mid\mid \texttt{fst}_s(\kappa_2) \\
\texttt{fst}_s(\kappa_1 + \kappa_2) &= \texttt{fst}_s(\kappa_1) + \texttt{fst}_s(\kappa_2) \\
\texttt{fst}_s(\kappa_1; \kappa_2) &= \begin{cases} \texttt{fst}_s(\kappa_1) & \text{si } \texttt{noinst}_{\mathit{Clocks}(s)}(\kappa_1) \\ \texttt{fst}_s(\kappa_1); \texttt{fst}_s(\kappa_2) & \text{sinon} \end{cases} \\
\texttt{fst}_s(\mu\phi.\, \kappa) &= \texttt{fst}_s(\kappa[\phi \leftarrow \mu\phi.\, \kappa]) \\
\texttt{fst}_s(\texttt{run}\ \kappa) &= \texttt{run}\ (\texttt{fst}_s(\kappa)) \\
\texttt{fst}_s(\texttt{domain}\ \kappa) &= \texttt{domain}\ (\texttt{fst}_s(\kappa))
\end{aligned}
$$

Le premier instant d'un comportement associé à une horloge ck est égal au comportement instantané 0 si l'horloge appartient à la pile s. Sinon, cela signifie que ck est plus rapide que $\texttt{top}(s)$ et il reste donc inchangé. Dans le cas de la séquence, on ne considère que le premier instant de κ_1 si on sait qu'il s'agit d'un comportement non instantané. Sinon il faut aussi ajouter le premier instant de κ_2. Dans le cas d'un comportement récursif, le premier instant du comportement n'est bien défini que si le comportement est réactif.

7.3 Système de types

L'abstraction d'un processus en un comportement est faite par un système de types-et-effets suivant l'approche de [ANN99] : le comportement d'un processus est son effet calculé par le système de types. On vérifie ensuite que le comportement obtenu est réactif. Nous allons présenter dans cette partie ce système de types-et-effets qui est proche du calcul d'horloges du chapitre 6 dans lequel on remplace les effets par des comportements. Nous verrons dans la partie 7.4 que si un programme est bien typé et que son comportement est réactif, alors le programme est réactif.

Types Les types de l'analyse de réactivité sont définis par :

$$rt ::= T \mid \alpha \mid \{ce\} \mid rt \times rt \mid (rt, rt)\texttt{event}\{ce\} \mid rt \to rt \mid rt\ \texttt{process}\{ce|\kappa\} \quad \text{(types)}$$
$$rs ::= rt \mid \forall\alpha.\, cs \mid \forall\phi.\, cs \quad \text{(schémas de type)}$$
$$\Gamma ::= \{x_1 \mapsto rs_1; \ldots; x_p \mapsto rs_p\} \quad \text{(environnements)}$$

Les types rt reprennent la même structure que dans le calcul d'horloges (voir page 96 du chapitre 6). Les deux différences sont que les fonctions $rt_1 \to rt_2$ n'ont plus d'effet et que le type rt $\texttt{process}\{ce|\kappa\}$ d'un processus est maintenant caractérisé par son type de retour rt, son horloge d'activation ce et son comportement κ. Les schémas de type rs sont maintenant paramétrés par les variables de type α et les variables de comportement ϕ. On étend la relation d'équivalence des comportements aux types : on a $rt_1 \equiv rt_2$ si les deux types sont égaux, hormis les comportements qu'ils contiennent qui sont équivalents. De même, un type est réactif s'il ne contient que des comportements réactifs.

Les jugements de typage sont de la forme :

$$\Gamma, ce \vdash e : rt \mid \kappa$$

qui signifie que dans l'environnement de typage Γ et l'horloge locale ce, l'expression e a le type rt et le comportement κ. On note $\Gamma, ce \vdash e : rt \mid \_$ lorsque l'on ignore le comportement de l'expression e car il n'est pas contraint et n'est pas utilisé dans le reste de la règle de typage.

Règles La figure 7.1 montre les règles du système de types. La gestion des types et des horloges est la même que dans le calcul d'horloges. On ne s'intéresse donc qu'au calcul des comportements :

- Dans le cas des expressions instantanées, c'est-à-dire telles que $0 \vdash e$ (voir partie 6.2), on ignore le comportement des sous-expressions, ce que l'on indique en notant _ leur comportement. On retourne toujours le comportement instantané 0. C'est par exemple le cas pour les variables, les constantes c ou encore les paires. Il faut bien noter que $0 \vdash e$ signifie que e est *nécessairement* instantanée, alors que $\Gamma, ce \vdash e : rt \mid 0$ signifie que e est *potentiellement* instantanée.
- On ne cherche pas à montrer la terminaison des fonctions, que l'on prend comme hypothèse. On peut le voir sur la règle APP : une fonction termine toujours instantanément. C'est pourquoi il n'y a pas de comportements associés aux fonctions, c'est-à-dire sur la flèche des fonctions, contrairement au calcul d'horloges et aux systèmes de types-et-effets traditionnels. Dans le cas de l'abstraction, on ignore le comportement du corps e_1 de la fonction.
- Les règles pour la définition et le lancement de processus sont les mêmes que pour le calcul d'horloges. On stocke le comportement κ_1 du corps e_1 du processus dans son type, puis on l'extrait à son lancement. On ajoute l'opérateur `run` au comportement du processus afin de garantir que le comportement associé à un processus récursif est toujours un comportement récursif. Supposons qu'on enlève cet opérateur du langage des comportements et considérons le processus $\texttt{rec}\ p = \texttt{process}\ (\texttt{run}\ p)$. En notant $\Gamma' = \Gamma[p \mapsto \beta\ \texttt{process}\{\gamma|0\}]$, on pourrait lui donner le comportement 0 et ainsi manquer la récursion instantanée :

$$\frac{\dfrac{\dfrac{\Gamma', ce \vdash p : \beta\ \texttt{process}\{\gamma|0\} \mid 0}{\Gamma', ce \vdash \texttt{run}\ p : \beta \mid 0}}{\Gamma', ce \vdash \texttt{process}\ (\texttt{run}\ p) : \beta\ \texttt{process}\{\gamma|0\} \mid 0}}{\Gamma, ce \vdash \texttt{rec}\ p = \texttt{process}\ (\texttt{run}\ p) : \beta\ \texttt{process}\{\gamma|0\} \mid 0}$$

  Grâce à l'opérateur `run`, le seul comportement que l'on peut donner à ce processus est $\mu\phi.\, \texttt{run}\ \phi$ qui est clairement non réactif. On note $\Gamma'' = \Gamma[p \mapsto \beta\ \texttt{process}\{\gamma|\mu\phi.\, \texttt{run}\ \phi\}]$:

$$\frac{\dfrac{\dfrac{\Gamma'', ce \vdash p : \beta\ \texttt{process}\{\gamma|\mu\phi.\, \texttt{run}\ \phi\} \mid 0}{\Gamma'', ce \vdash \texttt{run}\ p : \beta \mid \texttt{run}\ (\mu\phi.\, \texttt{run}\ \phi)}}{\Gamma'', ce \vdash \texttt{process}\ (\texttt{run}\ p) : \beta\ \texttt{process}\{\gamma|\mu\phi.\, \texttt{run}\ \phi\} \mid 0}}{\Gamma, ce \vdash \texttt{rec}\ p = \texttt{process}\ (\texttt{run}\ p) : \beta\ \texttt{process}\{\gamma|\mu\phi.\, \texttt{run}\ \phi\} \mid 0}$$

$$\frac{rt \leq \Gamma(x)}{\Gamma, ce \vdash x : rt \mid 0} \qquad \frac{rt \leq \Gamma_0(c)}{\Gamma, ce \vdash c : rt \mid 0} \qquad \frac{\Gamma, ce \vdash e_1 : rt_1 \mid \_ \qquad \Gamma, ce \vdash e_2 : rt_2 \mid \_}{\Gamma, ce \vdash (e_1, e_2) : rt_1 \times rt_2 \mid 0}$$

$$\frac{\Gamma[x \mapsto rt_2], ce \vdash e_1 : rt_1 \mid \_}{\Gamma, ce \vdash \lambda x.e_1 : rt_2 \to rt_1 \mid 0} \qquad (\textsc{App})\frac{\Gamma, ce \vdash e_1 : rt_2 \to rt_1 \mid \_ \quad \Gamma, ce \vdash e_2 : rt_2 \mid \_}{\Gamma, ce \vdash e_1\ e_2 : rt_1 \mid 0} \qquad \frac{\Gamma[x \mapsto rt], ce \vdash e_1 : rt \mid \_}{\Gamma, ce \vdash \texttt{rec}\ x = e_1 : rt \mid 0}$$

$$(\textsc{Process})\frac{\Gamma, ce' \vdash e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash \texttt{process}\ e_1 : rt\ \texttt{process}\{ce'|\kappa_1\} \mid 0} \qquad \frac{\Gamma, ce \vdash e_p : rt\ \texttt{process}\{ce|\kappa\} \mid \_}{\Gamma, ce \vdash \texttt{run}\ e_p : rt \mid \texttt{run}\ \kappa}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 : rt_1 \mid \kappa_1 \qquad \Gamma, ce \vdash e_2 : rt_2 \mid \kappa_2 \\ \Gamma[x_1 \mapsto gen(rt_1, e_1, \Gamma); x_2 \mapsto gen(rt_2, e_2, \Gamma)], ce \vdash e_3 : rt \mid \kappa_3\end{array}}{\Gamma, ce \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3 : rt \mid (\kappa_1 \mid\mid \kappa_2); \kappa_3}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_{ck} : \{ce'\} \mid \_ \qquad \Gamma, ce \vdash e_d : rt_2 \mid \_ \qquad \Gamma, ce \vdash e_g : rt_1 \to rt_2 \to rt_2 \mid \_ \\ \Gamma, ce \vdash e_{rck} : \{ce''\} \mid \_ \qquad \Gamma[x \mapsto (ct_1, ct_2)\texttt{event}\{ce'\}], ce \vdash e_1 : rt \mid \kappa_1\end{array}}{\Gamma, ce \vdash \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e_{ck}, \texttt{d} : e_d, \texttt{g} : e_g, \texttt{rck} : e_{rck})\ \texttt{in}\ e_1 : rt \mid 0; \kappa_1}$$

$$\frac{\Gamma, ce \vdash e_s : (rt_1, rt_2)\texttt{event}\{ce\} \mid \_ \qquad \Gamma, ce \vdash e_1 : rt \mid \kappa_1 \qquad \Gamma, ce \vdash e_2 : rt \mid \kappa_2}{\Gamma, ce \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : rt \mid \kappa_1 + (ce; \kappa_2)}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 : rt \mid \kappa_1 \\ \Gamma, ce \vdash e_s : (rt_1, rt_2)\texttt{event}\{ce'\} \mid \_ \\ \Gamma[x \mapsto rt_2], ce \vdash e_2 : rt \mid \kappa_2\end{array}}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2 : rt \mid \kappa_1 + (ce'; \kappa_2)} \qquad (\textsc{When})\frac{\Gamma, ce \vdash e_1 : rt \mid \kappa_1 \quad \Gamma, ce \vdash e_s : (rt_1, rt_2)\texttt{event}\{ce\} \mid \_}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{when}\ e_s : rt \mid 0; \kappa_1}$$

$$(\textsc{Pause})\frac{\Gamma, ce \vdash e_{ck} : \{ce'\} \mid \_}{\Gamma, ce \vdash \texttt{pause}\ e_{ck} : \texttt{unit} \mid ce'; 0} \qquad \frac{\Gamma, ce \vdash e_{ck} : \{ce'\} \mid \_}{\Gamma, ce \vdash \texttt{qpause}\ e_{ck} : \texttt{unit} \mid \top_{ck}; 0}$$

$$\frac{}{\Gamma, ce \vdash \texttt{local\_ck} : \{ce\} \mid 0}$$

$$(\textsc{DomainInf})\frac{\Gamma[x \mapsto \{\gamma\}], \gamma \vdash e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e_1 : rt \mid 0; \texttt{domain}\ (\kappa_1[\gamma \leftarrow 0])}$$

$$(\textsc{InInf})\frac{\Gamma, ck' \vdash e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash e_1\ \texttt{in}\ (ck', i/\infty) : rt \mid \texttt{domain}\ \left(\kappa_1[ck' \leftarrow 0]\right)}$$

$$(\textsc{DomainPeriod})\frac{\Gamma, ce \vdash e_b : \texttt{int} \mid \_ \qquad \Gamma[x \mapsto \{\gamma\}], \gamma \vdash e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1 : rt \mid 0; \texttt{domain}\ (\kappa_1[\gamma \leftarrow 0 \mid ce])}$$

$$(\textsc{InPeriod})\frac{\Gamma, ck' \vdash e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash e_1\ \texttt{in}\ (ck', i/j) : rt \mid \texttt{domain}\ (\kappa_1[\gamma \leftarrow 0 \mid ce])}$$

FIGURE 7.1 – Analyse de réactivité

- Le comportement de `let/and` traduit de façon immédiate la sémantique de cette construction. On exécute en parallèle e_1 et e_2 puis, lorsque les deux branches ont terminé, on exécute e_3.
- Dans le cas de la définition de signal, on type e_{ck} pour obtenir l'horloge ce' du signal, ainsi que les autres arguments dont on ignore le comportement puisqu'il s'agit d'expressions instantanées. La raison pour laquelle on renvoie un comportement $0;\kappa_1$ différent de celui du corps e_1 deviendra claire lorsque nous ferons la preuve de la sûreté de l'analyse dans la partie 7.4. L'idée est que le comportement d'une sous-expression doit toujours être plus petit que celui de l'expression complète.
- Dans le cas de l'expression `present` e `then` e_1 `else` e_2, la branche `then` est exécutée instantanément si le signal est présent, alors que la branche `else` est exécutée à l'instant suivant si le signal est absent. On peut voir que cela se traduit de façon évidente dans le comportement de cette construction.
- Dans le cas d'une préemption, on exécute soit le corps e_1, soit la continuation e_2 mais à l'instant suivant de l'horloge ce' du signal. C'est pourquoi le comportement de la construction est un choix entre le comportement κ_1 du corps et le comportement κ_2 de la continuation à l'instant suivant de ce.
- Comme pour la déclaration de signal, le comportement du `do/when` est plus grand que celui de son corps e_1 pour pouvoir mener à bien la preuve de sûreté (WHEN).
- Le comportement associé à l'expression `pause` ce' est $ce';0$, alors que celui associé à `qpause` ce' est toujours $\top_{ck};0$. Ceci montre bien la différence entre ces deux opérateurs : on peut boucler sur l'instant local d'un domaine réactif non borné, mais seulement si l'on utilise `qpause`. En effet, on obtient alors un comportement de la forme $\top_{ck}^{\infty}$ qui est bien réactif.
- Le type de l'expression `local_ck` est un type singleton associé à l'horloge locale ce, qui est à droite de l'environnement de typage Γ comme dans le calcul d'horloges. Son comportement est le comportement instantané 0 puisque l'expression s'évalue instantanément.
- On peut vérifier que les expressions dérivées du langage (voir figure 4.1) ont bien le comportement attendu. On peut prendre l'exemple de la séquence :

$$\frac{\Gamma, ce \vdash e_1 : rt_1 \mid \kappa_1 \qquad \Gamma, ce \vdash () : \texttt{unit} \mid 0 \qquad \Gamma, ce \vdash e_2 : rt_2 \mid \kappa_2}{\dfrac{\Gamma, ce \vdash \texttt{let \_} = e_1 \texttt{ and \_} = () \texttt{ in } e_2 : rt_2 \mid (\kappa_1 \mid\mid 0);\kappa_2}{\Gamma, ce \vdash e_1;e_2 : rt_2 \mid (\kappa_1 \mid\mid 0);\kappa_2}}$$

  On vérifie que $(\kappa_1 \mid\mid 0);\kappa_2 \equiv \kappa_1;\kappa_2$. De la même façon, on peut vérifier que $e_1 \mid\mid e_2$ a un comportement équivalent à $\kappa_1 \mid\mid \kappa_2$ ou encore que `await` $e_1(x)$ `in` e_2 a un comportement équivalent à $\top_{ck}^{\infty} + (ce';\kappa_2)$ où l'on note ce' l'horloge du signal e_1.
- On peut faire la même chose pour la construction `loop` qui est encodée par :

$$\texttt{loop } e \triangleq \texttt{run } ((\texttt{rec } loop = \lambda x.\texttt{process } (\texttt{run } x;\texttt{run } (loop\ x)))\ (\texttt{process } e))$$

  En appliquant les règles du système de types, on trouve que :

$$loop : \forall \phi, \gamma.\, \texttt{unit process}\{\gamma|\phi\} \to \texttt{unit process}\{\gamma|\mu\phi'.\,\texttt{run } \phi;\texttt{run } \phi'\}$$

  Si $\Gamma, ce \vdash e : rt \mid \kappa$, alors le comportement de cet encodage est : `run` $(\mu\phi'.\,\texttt{run } \kappa;\texttt{run } \phi')$. Il n'est pas équivalent à κ^{∞} au sens de la partie 7.2, mais il est réactif si et seulement si κ^{∞} l'est, puisque l'opérateur `run` n'influe pas sur la réactivité d'un comportement (voir la définition 10). Notre analyse traite donc le cas des boucles non instantanées sans avoir besoin de faire de cas particulier pour cette construction. Cela montre la généralité de l'approche choisie.

Cas des domaines réactifs On doit distinguer deux cas :

- Si le domaine est non borné, alors il faut s'assurer qu'il ne peut faire qu'un nombre borné d'instants locaux. Pour cela, on remplace l'horloge locale par le comportement instantané 0 (DOMAININF), ce qui traduit bien l'intuition que le domaine masque ses instants locaux. Si le comportement du corps contient un comportement récursif de la forme ck'^{∞}, alors on obtient

le comportement 0^∞ qui n'est pas réactif. On peut ainsi détecter que le domaine n'est potentiellement pas coopératif. On ajoute un comportement instantané avant celui du domaine (c'est-à-dire $0;\dots$) là encore pour la preuve de sûreté. On procède de la même façon pour le comportement d'un domaine en cours d'exécution e_1 `in` $(ck', i/\infty)$ (InInf), sauf que l'on remplace maintenant le nom ck' de l'horloge locale par 0.
- Si le domaine est borné, alors on sait déjà que le domaine est coopératif. Il faut tout de même traduire le comportement du domaine dans son horloge parente. On doit en effet savoir s'il boucle sur cette horloge ou s'il peut terminer instantanément. On utilise pour cela l'opération $\kappa[\gamma \leftarrow 0 \mid ce]$ (DomainPeriod). Elle substitue les occurrences de l'horloge locale γ par le comportement instantané 0, mais en prenant en compte que le comportement γ^∞ est en quelque sorte équivalent à ce^∞ puisque le domaine est borné (voir définition 11). On procède de la même manière pour le comportement d'un domaine en cours d'exécution e_1 `in` $(ck', i/j)$ (InPeriod).

7.4 Preuve de sûreté

Nous allons maintenant montrer la sûreté de notre analyse. Elle établit que si un programme est bien typé et si son comportement est réactif, alors le programme est réactif. Cela signifie qu'à chaque instant, le programme admet une dérivation finie dans la sémantique comportementale du chapitre 4 et qu'il se réécrit en un programme bien typé. L'intuition de la preuve est que le premier instant d'un comportement réactif est fini, sans aucune récursion. On montrera donc par induction sur la taille de ce premier instant du comportement la finitude de la dérivation. Pour les domaines réactifs non bornés, on montrera que la taille du premier instant du comportement du corps selon l'horloge parente du domaine décroit strictement à chaque instant local, ce qui prouve que le nombre d'instants locaux est borné pour chaque instant de l'horloge parente.

Commençons par énoncer la propriété essentielle des comportements réactifs :

Lemme 7.2. *Si un comportement est réactif, alors son premier instant est fini :*

$$Clocks(s), \emptyset \vdash \kappa \quad \Rightarrow \quad |\mathtt{fst}_s(\kappa)| < \infty$$

Démonstration. Par induction sur la dérivation de $Clocks(s), \emptyset \vdash \kappa$. Pour que la taille du comportement soit finie, il faut montrer que la variable ϕ de récursion de chaque comportement récursif $\mu\phi.\,\kappa$ n'apparait pas dans le premier instant du corps κ, ce qui est le but de la définition de la réactivité des comportements (définition 10). □

On peut montrer deux lemmes un peu techniques qui relient la taille du premier instant du corps d'un domaine dans l'horloge locale avec celle du domaine dans son horloge parente :

Lemme 7.3. *Soit s une pile et $ck', ck'' \notin Clocks(s)$. Alors :*
- *Si* $|\mathtt{fst}_s(\mathtt{domain}\ (\kappa[ck' \leftarrow 0]))| < \infty$
  alors $|\mathtt{fst}_{s::(ck',i)}(\kappa)| < |\mathtt{fst}_s(\mathtt{domain}\ (\kappa[ck' \leftarrow 0]))| < \infty$
- *Si* $|\mathtt{fst}_s(\mathtt{domain}\ (\kappa[ck' \leftarrow 0 \mid ck'']))| < \infty$
  alors $|\mathtt{fst}_{s::(ck',i)}(\kappa)| < |\mathtt{fst}_s(\mathtt{domain}\ (\kappa[ck' \leftarrow 0 \mid ck'']))| < \infty$

Démonstration. Le fait de remplacer un comportement 0 par une horloge ck' ne change rien si l'horloge n'appartient pas à $Clocks(s)$, puisque l'on reste dans le deuxième cas de la définition 13. Pour le deuxième point, on peut aussi remplacer ck' par ck'', mais là aussi cela ne change rien puisque les deux horloges n'appartiennent pas à $Clocks(s)$. La présence de l'opérateur `domain` garantit que la taille décroit strictement. □

Lemme 7.4. *Soit s une pile et $ck', ck'' \notin Clocks(s)$. Alors :*
- *Si* $|\mathtt{fst}_s(\kappa')| < |\mathtt{fst}_s(\kappa)|$ *alors* $|\mathtt{fst}_s(\mathtt{domain}\ \kappa'[ck' \leftarrow 0])| < |\mathtt{fst}_s(\mathtt{domain}\ \kappa[ck' \leftarrow 0])|$.
- *Si* $|\mathtt{fst}_s(\kappa')| < |\mathtt{fst}_s(\kappa)|$
  alors $|\mathtt{fst}_s(\mathtt{domain}\ \kappa'[ck' \leftarrow 0 \mid ck''])| < |\mathtt{fst}_s(\mathtt{domain}\ \kappa[ck' \leftarrow 0 \mid ck''])|$.

On a les mêmes relations avec $\leq$.

Avant de commencer la preuve proprement dite, il nous faut traduire de façon formelle notre hypothèse que les fonctions terminent. Cela revient à supposer l'existence d'une dérivation finie dans la sémantique comportementale pour toute expression instantanée :

Hypothèse 7.5 (Les fonctions terminent)**.** *Pour tout expression surement instantanée e, c'est-à-dire telle que $0 \vdash e$, il existe une dérivation Π et une valeur v telle que :*

$$\frac{\Pi}{N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},0} v}$$

Nous allons d'abord montrer la sûreté de notre définition de comportement non instantané :

Lemme 7.6. *Une expression dont le comportement est non instantané ne se réduit jamais instantanément :*

$$\left(N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e' \ \wedge \ \Gamma, \mathtt{top}(s) \vdash e : rt \mid \kappa \ \wedge \ \mathtt{noinst}_s(\kappa)\right) \Rightarrow k \neq 0$$

Démonstration. Par induction sur la dérivation de la sémantique comportementale. □

Nous allons maintenant énoncer la sûreté de l'analyse : un programme bien typé avec un comportement réactif est réactif, c'est-à-dire que sa réaction est décrite par une dérivation finie dans la sémantique comportementale.

Théorème 7.7 (Sureté de l'analyse de réactivité)**.** *Si $\Gamma, \top_{ck} \vdash e : ct \mid cf$ et $\Gamma, \top_{ck} \vdash e : rt \mid \kappa$ avec $\{\top_{ck}\}, [] \vdash \kappa$ et si on suppose que les fonctions terminent (hypothèse 7.5), alors il existe e' telle que $N, (\top_{ck}, i) \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'$.*

Démonstration. La preuve de sûreté du calcul d'horloges montre que l'on aura aucune erreur d'exécution, c'est-à-dire que les conditions sur les horloges des signaux sont toujours vérifiées. On ne s'intéresse donc ici qu'à la finitude de la dérivation dans la sémantique comportementale. On fait une récurrence sur la taille du premier instant du comportement κ. On doit utiliser une hypothèse de récurrence un peu plus générale, pour pouvoir faire la récurrence dans n'importe quelle horloge locale :

> Pour toute pile s, si $\Gamma, \mathtt{top}(s) \vdash e : ct \mid cf$ et $\Gamma, \mathtt{top}(s) \vdash e : rt \mid \kappa$ avec κ réactif dans la pile s, alors il existe e' telle que $N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'$.

Nous allons d'abord justifier la correction dans le cas des expressions de base :

Cas $e_1\ e_2$ et $\mathtt{rec}\ x = e_1$. On applique l'hypothèse 7.5 puisque l'on a $0 \vdash e$.

Cas $\mathtt{run}\ e_p$. On sait que $0 \vdash e_p$ et que $\mathtt{run}\ e_p$ est bien typé, donc il existe une dérivation Π telle que :

$$\frac{\Pi}{N_1, s \vdash e_p \xrightarrow[\mathcal{S}]{\mathcal{E},0} \mathtt{process}\ e_1}$$

L'expression e_p se réécrit en une valeur par l'hypothèse 7.5, qui est nécessairement de la forme $\mathtt{process}\ e_1$ car il s'agit de la seule valeur qui peut avoir le type $rt\ \mathtt{process}\{ce|\kappa\}$. Le calcul d'horloges garantit que l'effet de $\mathtt{process}\ e_1$ est nul, dont on déduit que $N_1 = \emptyset$ et $\mathcal{E} = \emptyset$. On peut ensuite construire la dérivation de type suivante :

$$\frac{\dfrac{\Gamma, \mathtt{top}(s) \vdash e_1 : rt \mid \kappa}{\Gamma, \mathtt{top}(s) \vdash \mathtt{process}\ e_1 : rt\ \mathtt{process}\{ce|\kappa\} \mid 0}}{\Gamma, \mathtt{top}(s) \vdash \mathtt{run}\ (\mathtt{process}\ e_1) : rt \mid \mathtt{run}\ \kappa}$$

On peut appliquer l'hypothèse d'induction puisque le premier instant du comportement de e_1, c'est-à-dire $\texttt{fst}_s(\kappa)$, est plus petit que le premier instant du comportement de $\texttt{run}\ (\texttt{process}\ e_1)$, qui est aussi celui de $\texttt{run}\ e_p$, puisque e_p et $\texttt{process}\ e_1$ ont le même type. En effet, on a $\texttt{fst}_s(\texttt{run}\ \kappa) = \texttt{run}\ (\texttt{fst}_s(\kappa))$.
L'hypothèse d'induction nous permet de déduire que :

$$\frac{\Pi_1}{N_2, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1, k} e_1'}$$

ce qui nous permet de construire la dérivation complète de $\texttt{run}\ e_p$:

$$\frac{\dfrac{\Pi}{\emptyset, s \vdash e_p \xrightarrow[\mathcal{S}]{\emptyset, 0} \texttt{process}\ e_1} \qquad \dfrac{\Pi_1}{N_2, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}, k} e_1'}}{N_1 \cdot N_2, s \vdash \texttt{run}\ e_p \xrightarrow[\mathcal{S}]{\mathcal{E}, k} e_1'}$$

Cas $\texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2$**.** : Comme dans le cas précédent, on a :

$$\frac{\Pi}{\emptyset, s \vdash e_s \xrightarrow[\mathcal{S}]{\emptyset, 0} n^{ck}}$$

La règle de typage donne :

$$\frac{\begin{array}{c}\Gamma, \texttt{top}(s) \vdash e_s : (rt_1, rt_2)\texttt{event}\{\texttt{top}(s)\} \mid \_ \\ \Gamma, \texttt{top}(s) \vdash e_1 : rt \mid \kappa_1 \qquad \Gamma, \texttt{top}(s) \vdash e_2 : rt \mid \kappa_2\end{array}}{\Gamma, \texttt{top}(s) \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : rt \mid \kappa_1 + (\texttt{top}(s); \kappa_2)}$$

On a deux cas suivant le statut de n :
– Si $n \in \mathcal{S}(s)$: on peut remarquer que

$$\texttt{fst}_s(\kappa_1 + (\texttt{top}(s); \kappa_2)) = \texttt{fst}_s(\kappa_1) + 0$$

On peut donc appliquer l'hypothèse d'induction sur le premier instant du comportement de e_1 et conclure.
– Si $n \notin \mathcal{S}(s)$: la dérivation est finie.

Cas $\texttt{do}\ e_1\ \texttt{when}\ e_s$**.** On peut appliquer le même raisonnement. On peut remarquer que le comportement associé à cette construction (c'est-à-dire $0; \kappa$) n'est pas égal à celui du corps (c'est-à-dire κ) comme on aurait pu s'y attendre afin que l'on puisse appliquer l'hypothèse d'induction.

Cas $\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3$**.** Lorsque l'on calcule le premier instant d'une séquence, deux cas peuvent se produire :
– Si $\texttt{noinst}_s(\kappa_1 \mid\mid \kappa_2)$, alors on a $\texttt{fst}_s(\kappa) = \texttt{fst}_s(\kappa_1) \mid\mid \texttt{fst}_s(\kappa_2)$. Puisque $\texttt{noinst}_s(\kappa_1 \mid\mid \kappa_2)$, on a soit $\texttt{noinst}_s(\kappa_1)$, qui implique que $k_1 \neq 0$ en utilisant le lemme 7.6, soit $\texttt{noinst}_s(\kappa_2)$, qui implique que $k_2 \neq 0$. On est donc sûr que $\min_s(k_1, k_2) \neq 0$. On peut appliquer l'hypothèse d'induction sur e_1 et e_2, puis construire la dérivation complète avec la règle LETPAR.
– Sinon, on a $\texttt{fst}_s(\kappa) = (\texttt{fst}_s(\kappa_1) \mid\mid \texttt{fst}_s(\kappa_2)); \texttt{fst}_s(\kappa_3)$. On peut donc appliquer l'hypothèse d'induction sur e_1, e_2 et e_3 et utiliser une des deux règles de la sémantique comportementale (LETDONE ou LETPAR) selon leurs statuts de retour.

Cas $\texttt{pause}\ e_{ck}$ **et** $\texttt{qpause}\ e_{ck}$**.** On peut appliquer l'hypothèse 7.5 pour montrer que la dérivation de e_{ck} est finie et conclure simplement.

Intéressons nous maintenant au cas des domaines réactifs, c'est-à-dire de $e_1\ \texttt{in}\ (ck', i/j)$. On note κ_1 le comportement de e_1. Si $0 \vdash e_1$, alors on conclut simplement en utilisant l'hypothèse 7.5. Sinon, on a deux cas :

Cas $e_1 = \infty$**.** On a alors $\kappa = \texttt{domain}\ \left(\kappa_1[ck' \leftarrow 0]\right)$. On a donc $|\texttt{fst}_{s'}(\kappa_1)| < |\texttt{fst}_s(\kappa)|$ (lemme 7.3) ce qui nous permet d'appliquer l'hypothèse d'induction sur e_1. On obtient donc une dérivation :

$$\frac{\Pi_1}{N_1, s \vdash e_1 \xrightarrow[\mathcal{S}]{\mathcal{E}_1,k} e_1'}$$

On raisonne ensuite selon la valeur de k :
- Si $k = 0$, alors on peut construire la dérivation de e en utilisant la règle TERM.
- Si $k = ck'$, on peut appliquer le lemme 7.9 que l'on montrera un peu plus tard pour montrer que le comportement κ_1' de e_1' vérifie $|\texttt{fst}_s(\kappa_1')| < |\texttt{fst}_s(\kappa_1)|$. On a bien en effet $\texttt{fst}_{s_{\downarrow ck}}(\kappa_1) < \infty$ et $\texttt{eoi}_{s',\mathcal{S}}(ck')$. On a donc en utilisant le lemme 7.4 :

  $$|\texttt{fst}_s(\texttt{domain}\ \left(\kappa_1'[ck' \leftarrow 0]\right))| < |\texttt{fst}_s(\texttt{domain}\ \left(\kappa_1[ck' \leftarrow 0]\right))|$$

  L'expression e_1' `in` $(ck', i+1/j)$ a le comportement $\texttt{domain}\ \left(\kappa_1'[ck' \leftarrow 0]\right)$, donc on peut lui appliquer l'hypothèse d'induction car son premier instant est plus petit. On conclut en utilisant la règle LOCALEOI. On sait en effet que la condition $i < j - 1$ est toujours vérifiée comme $j = \infty$.
- Si $k = ck'' \neq ck'$, on peut conclure en utilisant la règle PARENTEOI.

On assure donc que le nombre d'instants locaux est fini car la taille du premier instant du comportement du domaine décroit strictement à chaque instant local, ce que montre le lemme 7.9.

Cas $e_1 \neq \infty$**.** On peut montrer que $|\texttt{fst}_{s'}(\kappa_1)| < |\texttt{fst}_s(\kappa)|$ par le lemme 7.3. On peut donc de la même façon appliquer l'hypothèse d'induction sur e_1 qui réagit en e_2. On a alors plusieurs cas selon les valeur de k et i :
- Si $k = 0$, alors on peut construire la dérivation de e en utilisant la règle TERM.
- Si $k = ck'' \neq ck'$, on peut conclure en utilisant la règle PARENTEOI.
- Si $i = j - 1$, alors on utilise la règle COUNTEREOI.
- Sinon, on recommence le même raisonnement avec e_2.

On sait que notre raisonnement va terminer car le nombre d'instants locaux est borné par j, donc on arrive forcément au cas $i = j - 1$.

□

Dans le cas d'un domaine réactif non borné, on a eu besoin de montrer que la taille du premier instant (sur l'horloge parente) du corps du domaine décroissait strictement à chaque instant local. On va tout d'abord montrer que cette taille décroit non strictement, avec un lemme rappelant la préservation du typage.

Lemme 7.8. *Pour toute pile s avec $\texttt{top}(s) \prec_s ck$, si $N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E},k} e'$ et $\Gamma, \texttt{top}(s) \vdash e : rt \mid \kappa$ avec $|\texttt{fst}_{s_{\downarrow ck}}(\kappa)| < \infty$ et ~~$0 \vdash e$~~ alors $\Gamma, \texttt{top}(s) \vdash e' : rt \mid \kappa'$ avec $|\texttt{fst}_{s_{\downarrow ck}}(\kappa)| \leq |\texttt{fst}_{s_{\downarrow ck}}(\kappa')|$.*

Démonstration. Par récurrence sur la dérivation de la sémantique comportementale de e. On ne traite pas le cas des expressions instantanées puisque les règles de typage ne garantissent pas cette propriété dans leur cas. On utilise à la place l'hypothèse 7.5 dans la preuve de sûreté de l'analyse.

Cas e_1 `in` $(ck', i/j)$ **avec** $0 \vdash e_1$**.** On ne peut pas appliquer l'hypothèse d'induction sur e_1 dans ce cas. Mais l'hypothèse 7.5 nous dit que $e' = v$ et donc $\kappa' = 0$ (en utilisant la règle TERM). On a donc $|\texttt{fst}_{s_{\downarrow ck}}(\kappa')| = 1$ ce qui permet de conclure puisque la taille d'un comportement est toujours supérieure à 1.

Cas e_1 `in` $(ck', i/\infty)$**.** On peut maintenant appliquer l'hypothèse d'induction sur e_1 de comportement κ_1 tel que $\kappa = \kappa_1[ck' \leftarrow 0]$, qui réagit en e_1' de comportement κ_1'. En effet, on a $|\texttt{fst}_{s'_{\downarrow ck}}(\kappa_1)| < \infty$ en appliquant le lemme 7.3. On a donc $|\texttt{fst}_{s'_{\downarrow ck}}(\kappa_1')| \leq |\texttt{fst}_{s'_{\downarrow ck}}(\kappa_1)|$ par le lemme 7.4. En notant $\kappa' = \texttt{domain}\ \left(\kappa_1'[ck' \leftarrow 0]\right)$, on en déduit que $|\texttt{fst}_{s_{\downarrow ck}}(\kappa')| \leq |\texttt{fst}_{s_{\downarrow ck}}(\kappa)|$. Cela permet de conclure dans le cas des règles TERM et PARENTEOI (la règle COUNTEREOI

ne peut pas être utilisée car $j = \infty$). Dans le cas de la règle LOCALEOI, il faut à nouveau utiliser l'hypothèse d'induction sur e'_1 `in` $(ck', i+1/\infty)$ de comportement κ' vérifiant bien $|\mathtt{fst}_{s_{\restriction ck}}(\kappa')| < \infty$ et qui est bien plus petit. On obtient alors $|\mathtt{fst}_{s_{\restriction ck}}(\kappa'')| \leq |\mathtt{fst}_{s_{\restriction ck}}(\kappa')|$ et on conclut par transitivité de $\leq$.

Cas e_1 `in` $(ck', i/j)$**.** On a maintenant $\kappa = \kappa_1[ck' \leftarrow 0 \mid \mathtt{top}(s)]$. On peut appliquer le lemme 7.3 et raisonner de la même façon.

□

On peut maintenant montrer le lemme qui nous intéresse, c'est-à-dire que le comportement décroit strictement à chaque instant local :

Lemme 7.9. *Pour toute pile s et* $\mathtt{top}(s) \preceq_s ck'' \prec_s ck$*, si* $N, s \vdash e \xrightarrow[\mathcal{S}]{\mathcal{E}, ck''} e'$ *et* $\Gamma, \mathtt{top}(s) \vdash e : rt \mid \kappa$ *avec* $|\mathtt{fst}_{s_{\restriction ck}}(\kappa)| < \infty$ *et* ~~$\emptyset \vdash e$~~ *et* $\mathtt{eoi}_{s,\mathcal{S}}(ck'')$*, alors* $\Gamma, \mathtt{top}(s) \vdash e' : rt \mid \kappa'$ *avec* $|\mathtt{fst}_{s_{\restriction ck}}(\kappa')| < |\mathtt{fst}_{s_{\restriction ck}}(\kappa)|$.

Démonstration. Par récurrence sur la dérivation de la sémantique comportementale de e. L'hypothèse $\mathtt{eoi}_{s,\mathcal{S}}(ck)$ assure que l'expression e réagit en une expression e' différente, ce qui est nécessaire si on veut que la taille du comportement décroisse.

Cas `let` $x_1 = e_1$ `and` $x_2 = e_2$ `in` e_3**.** On a alors deux cas possibles :
- Soit $k = \min_s(k_1, k_2) = 0$. Alors on applique l'induction sur e_3 puisque son statut est bien égal à ck''.
- Soit $\min_s(k_1, k_2) = ck''$. Alors on sait que k_1 ou k_2 est égal à ck''. Supposons qu'il s'agisse de k_1. On peut donc appliquer l'hypothèse d'induction sur e_1 et on obtient un nouveau comportement κ'_1 tel que $|\mathtt{fst}_{s_{\restriction ck}}(\kappa'_1)| < |\mathtt{fst}_{s_{\restriction ck}}(\kappa_1)|$. Il faut utiliser le lemme 7.8 pour e_2 qui nous donne $|\mathtt{fst}_{s_{\restriction ck}}(\kappa'_2)| \leq |\mathtt{fst}_{s_{\restriction ck}}(\kappa_2)|$. On en déduit donc que $|\mathtt{fst}_{s_{\restriction ck}}(\kappa')| < |\mathtt{fst}_{s_{\restriction ck}}(\kappa)|$ puisque $|\mathtt{fst}_{s_{\restriction ck}}(\kappa')| = |\mathtt{fst}_{s_{\restriction ck}}(\kappa'_1)| + |\mathtt{fst}_{s_{\restriction ck}}(\kappa'_2)| + 1 + |\mathtt{fst}_{s_{\restriction ck}}(\kappa_3)|$ (idem pour κ).

Cas `pause` ck''**.** Comme $\mathtt{eoi}_{s,\mathcal{S}}(ck'')$, on a $e' = ()$. On a donc bien :

$$|\mathtt{fst}_{s_{\restriction ck}}(\kappa')| = |0| = 1 < |\mathtt{fst}_{s_{\restriction ck}}(ck'; 0)| = 3$$

Cas e_1 `in` $(ck', i/j)$ **avec** $0 \vdash e_1$**.** On fait le même raisonnement que dans le lemme précédent et on obtient bien $|\mathtt{fst}_{s_{\restriction ck}}(\kappa')| < |\mathtt{fst}_{s_{\restriction ck}}(\kappa)|$.

Cas e_1 `in` $(ck', i/\infty)$**.** On a plusieurs cas selon la règle de la sémantique que l'on utilise :
- Règle TERM : Impossible car le statut est égal à ck.
- Règle COUNTEREOI : Impossible car $j = \infty$.
- Règle LOCALEOI : On fait l'induction sur e'_1 `in` $(ck', i+1/j)$ qui a le même statut de retour, ce qui nous donne $|\mathtt{fst}_{s_{\restriction ck}}(\kappa')| < |\mathtt{fst}_{s_{\restriction ck}}(\kappa'_1[ck' \leftarrow 0])|$. On utilise en outre le lemme 7.8 pour montrer que $|\mathtt{fst}_{s_{\restriction ck}}(\kappa'_1[ck' \leftarrow 0])| \leq |\mathtt{fst}_{s_{\restriction ck}}(\kappa_1[ck' \leftarrow 0])|$ ce qui nous permet de déduire que $|\mathtt{fst}_{s_{\restriction ck'}}(\kappa')| < |\mathtt{fst}_{s_{\restriction ck'}}(\kappa)|$.
- Règle PARENTEOI : On peut faire l'induction sur e_1 qui a le même statut de retour. On a bien $\mathtt{eoi}_{s::(ck',i),\mathcal{S}}(ck'')$ car on est dans le dernier instant de l'horloge ck' pour l'instant courant de l'horloge $\mathtt{top}(s)$, dont on sait qu'il s'agit du dernier instant pour l'instant courant de ck'' puisque $\mathtt{eoi}_{s,\mathcal{S}}(ck'')$. On en déduit le résultat attendu.

Cas e_1 `in` $(ck', i/j)$**.** Comme pour le lemme précédent, on peut procéder de la même façon que pour le cas $j = \infty$. Il reste simplement à traiter le cas de la règle COUNTEREOI. On a deux cas :
- Soit $k = ck''$: Alors on applique l'hypothèse d'induction sur e_1 et on conclut simplement comme dans le cas de PARENTEOI.
- Soit $k = ck'$: Alors on applique l'hypothèse d'induction sur e_1 en prenant $ck'' = ck'$, ce que nous permet le lemme 7.3. On peut alors conclure.

□

7.5 Discussion

Simplification des comportements

Le comportement que le système de types associe à chaque expression est très grand. Par exemple, le comportement du processus `timer` (page 110) est $((0 \mid\mid 0); (0 + (0; 0)))^\infty$. Ce comportement est inutilement détaillé et est quasiment aussi grand que le code source du programme. On remarque pourtant qu'il est équivalent à 0^∞, ce qui est la seule information dont on a vraiment besoin pour juger de sa réactivité.

On voudrait pouvoir utiliser la relation d'équivalence sur les comportements, définie dans la partie 7.2, pour simplifier les comportements obtenus par notre système de types. On peut exprimer cela sous la forme d'une nouvelle règle de typage, qui permet de simplifier le comportement à tout moment. On note $\vdash_S$ le système de types étendu avec la nouvelle règle, qui est celui qui est implémenté dans le compilateur :

$$\text{(EQUIV)}\ \frac{\Gamma, ce \vdash_\text{S} e : rt \mid \kappa_1 \qquad \kappa_1 \equiv \kappa_2}{\Gamma, ce \vdash_\text{S} e : rt \mid \kappa_2}$$

Pour pouvoir utiliser cette règle, on doit montrer qu'elle est *admissible*, c'est-à-dire qu'elle ne change pas l'ensemble des programmes acceptés par le système de types. Ceci est garanti par le fait que l'équivalence préserve la réactivité (propriété 7.1), qui est la propriété requise par la preuve de sûreté. La preuve requiert également que le comportement des sous-expressions soit plus petit, mais ceci est une propriété du système de types, pas d'un programme en particulier.

Propriété 7.10. *Si* $\Gamma, ce \vdash_S e : rt \mid \kappa$*, alors* $\Gamma, ce \vdash e : rt \mid \kappa'$ *avec* $\kappa \equiv \kappa'$ *et* $rt \equiv rt'$.

Démonstration. Immédiat par induction sur les règles de typage. □

La conséquence de cette propriété est la sûreté du système de types étendu. On peut ensuite simplifier certaines règles du système de type original en les combinant avec la règle EQUIV. Ainsi, les règles WHEN et PAUSE deviennent :

$$\text{(WHEN')}\ \frac{\Gamma, ce \vdash_\text{S} e_1 : rt \mid \kappa_1 \qquad \Gamma, ce \vdash_\text{S} e_s : (rt_1, rt_2)\texttt{event}\{ce\} \mid \_}{\Gamma, ce \vdash_\text{S} \texttt{do}\ e_1\ \texttt{when}\ e_s : rt \mid \kappa_1}$$

$$\text{(PAUSE')}\ \frac{\Gamma, ce \vdash_\text{S} e_{ck} : \{ce'\} \mid \_}{\Gamma, ce \vdash_\text{S} \texttt{pause}\ e_{ck} : \texttt{unit} \mid ce'}$$

On peut également enlever l'opérateur `domain` du langage des comportements, puisqu'il ne sert qu'à garantir la décroissance des comportements. On obtient donc par exemple :

$$\text{(DOMAININF')}\ \frac{\Gamma[x \mapsto \{\gamma\}], \gamma \vdash_\text{S} e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash_\text{S} \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e_1 : rt \mid \kappa_1[\gamma \leftarrow 0]}$$

$$\text{(DOMAINPERIOD')}\ \frac{\Gamma, ce \vdash_\text{S} e_b : \texttt{int} \mid \_ \qquad \Gamma[x \mapsto \{\gamma\}], \gamma \vdash_\text{S} e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash_\text{S} \texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1 : rt \mid \kappa_1[\gamma \leftarrow 0 \mid ce]}$$

Exemples

L'utilisation d'un système de types pour notre analyse permet de gérer les cas d'alias :

```
let rec process p =
  let q = p in
  run q
```

```
let rec process p =
  let q = (fun x -> x) p in
  run q
```

```
val p : 'a process['c0 | rec 'r. run 'r]
Warning: This expression may produce an instantaneous recursion.
```

Il faut appeler le compilateur avec l'option `-dreactivity` pour afficher les comportements calculés par l'analyse de réactivité. Ils sont alors affichés entre crochets après l'horloge d'activation et l'effet du processus calculés par le calcul d'horloges (qui sont entre accolades). Nous n'affichons ici que l'horloge d'activation (dans l'exemple `'c0`) et le comportement des processus (dans l'exemple `rec 'r. run 'r`) entre crochets pour simplifier les signatures.

Dans les deux exemples, le processus `q` a le même type que `p` et donc le même comportement. La récursion instantanée est donc détectée.

L'analyse peut aussi gérer le cas des combinateurs, comme le processus par_comb (page 113) :

exemples/ch7/par_comb.rml

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
```

```
val par_comb : 'a process['c0 | 'r1] -> 'b process ['c0 | 'r2] ->
               unit process['c0 | rec 'r3. run 'r1  ||  run 'r2 ; run 'r3]
```

L'analyse n'affiche pas d'avertissement au moment de la définition du processus puisque l'on ne peut pas encore décider s'il est réactif. En effet, la composition parallèle synchrone termine quand les deux branches ont terminé. La boucle est donc non instantanée si `q1` ou `q2` est non instantané. Plus formellement, le comportement obtenu est réactif car les variables de comportement sont considérées comme non instantanées (voir la définition 9). On vérifie ensuite la réactivité au moment de l'instanciation du combinateur. Si on instancie `par_comb` avec un processus instantané et un non instantané, alors on obtient bien un processus réactif (on note `{'c}` le comportement associé à l'horloge `'c`) :

```
let process p1 = run par_comb (process ()) (process (pause))
val p1 : unit process['c0 | rec 'r. run 0  ||  run {'c0} ; run 'r]
```

Si les deux processus sont instantanés, la boucle devient instantanée. Notre analyse détecte bien ce problème de réactivité :

```
let process p2 = run par_comb (process ()) (process ())
val p2 : unit process['c0 | rec 'r. run 0  ||  run 0 ;  run 'r]
Warning: This expression may produce an instantaneous recursion.
```

On peut considérer un exemple plus complexe d'utilisation de fonctions d'ordre supérieur et de processus. On définit une fonction `higher_order` qui prend en entrée un combinateur `f`. Elle crée ensuite un processus récursif qui applique `f` à lui-même puis lance le résultat :

exemples/ch7/higher_order.rml

```
let higher_order f =
 let rec process p =
   let q = f p in
   run q
 in
 p
```

```
val higher_order : ('a process['c0 | run 'r1] -> 'a process['c0 | 'r1])
                        -> 'a process['c0 | run 'r1]
```

Si on instancie cette fonction avec un combinateur qui attend un instant avant de lancer son argument, on obtient un processus réactif :

```
let process good = run higher_order (proc x -> pause; run x)
val good : 'a process['c0 | run (run (rec 'r1. {'c0} ; run (run 'r1)))]
```

Ce n'est plus le cas si le combinateur appelle son argument instantanément. L'analyse détecte la récursion instantanée :

```
let process pb = run higher_order (proc x -> run x)
val pb : 'a process['c0 | run (run (rec 'r2. run (run 'r2)))]
Warning: This expression may produce an instantaneous recursion.
```

Un autre exemple intéressant est un opérateur de point-fixe. Il s'agit d'une fonction qui attend une continuation et l'applique en se donnant elle-même comme continuation. Cet opérateur de point-fixe peut être utilisé pour créer un processus récursif, dont la réactivité est bien vérifiée par notre système de types :

exemples/ch7/fix.rml

```
let rec fix f x = f (fix f) x
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

let process fix_main =
  let process p k v =
    print_int v; print_newline ();
    run (k (v+1))
  in
  run (fix p 0)
val main : 'a process['c0 | run rec 'r0. run 'r0]
Warning: This expression may produce an instantaneous recursion.
```

Exemples avec domaines réactifs

L'analyse de réactivité gère également le cas des domaines réactifs. Si un domaine réactif non borné boucle sur l'horloge locale, comme dans le cas du processus `nonreactive_domain` (page 111), alors l'analyse affiche un avertissement :

exemples/ch3/nonreactive_domain.rml

```
let process nonreactive_domain =
  domain ck do
    loop pause ck end
  done
val nonreactive_domain : unit process['c0 | rec 'r1. (0 ; run 'r1)]
Warning: This reactive domain may not cooperate.
```

Le mot-clé `domain` n'apparait pas dans le comportement du processus car il n'est plus utile dans le système de types avec simplification et ne sert qu'à la preuve de sûreté. Si on ajoute une période au domaine réactif, alors ce programme devient coopératif :

exemples/ch7/periodic_domain.rml

```
let process periodic_domain =
  domain ck by 2 do
    loop pause ck end
  done
val periodic_domain : unit process['c0 | rec 'r1. ({'c0} ;  run 'r1)]
```

On peut remarquer que l'on obtient alors un processus qui boucle sur son horloge d'activation. L'analyse affiche un avertissement si on lance ce processus à l'intérieur d'un domaine réactif non borné, puisque celui-ci devient non coopératif :

exemple/ch7/nested_domains.rml
```
let process nested_domain =
  domain ck2 do
    run periodic_domain
  done
```
```
val nested_domain : unit process['c0 | run (rec 'r1. (0 ; run 'r1))]
Warning: This reactive domain may not cooperate.
```

Une autre façon de corriger le processus nonreactive_domain est d'utiliser l'opérateur **quiet pause**. On obtient alors un domaine coopératif puisque le comportement de qpause ck est toujours $\top_{ck}$:

exemples/ch7/qpause_domain.rml
```
let process qpause_domain =
  domain ck do
    loop quiet pause ck end
  done
```
```
val qpause_domain : unit process['c0 | rec 'r1. ({?topck} ; run 'r1)]
```

Notre analyse détecte également les domaines réactifs qui peuvent terminer instantanément et rendre une boucle infinie instantanée, comme le processus instantaneous_domain (page 112) :

exemples/ch7/instantaneous_domain.rml
```
let process instantaneous_domain =
  loop
    domain ck do
      pause ck; pause ck
    done
  end
```
```
val instantaneous_domain : unit process {'c0|rec 'r0. (0 ; 0 ; run 'r0)}
Warning: This expression may be an instantaneous loop.
```

Le résultat est le même dans le cas où l'on force le domaine à avoir une période égale à 2 :

exemples/ch7/instantaneous_domain_period.rml
```
let process instantaneous_domain_period =
  loop
    domain ck by 2 do
      pause ck; pause ck
    done
  end
```
```
val instantaneous_domain_period : unit process {'c0|rec 'r0. (0 ; 0 ; run 'r0)}
Warning: This expression may be an instantaneous loop.
```

Puisque le domaine réactif n'exécute que deux instants locaux par instant de son horloge parente, son exécution prend un instant de son horloge parente, ce qui rend la boucle non instantanée. Notre analyse affiche tout de même un avertissement car elle ignore la valeur de la période, qui peut être une expression quelconque. En effet, la boucle serait bien instantanée si on avait par exemple choisi une période de 3.

Implémentation

Comme pour le calcul d'horloges, l'implémentation de l'analyse de réactivité ne pose pas de difficulté. Nous avons choisi de calculer les comportements pendant le calcul d'horloges. Les définitions de la partie 7.2 (page 114) se traduisent en un algorithme pour vérifier la réactivité d'un comportement, de complexité polynomiale en la taille du comportement. Cette vérification se fait après le calcul d'horloges. La fonction de substitution conditionnelle $\kappa[ck' \leftarrow 0 \mid ck]$ est un peu plus complexe. Le principe est de parcourir le comportement en mémorisant la liste des instances de ck' qui peuvent rendre le comportement non réactif et celles qui sont sûres. Lorsque l'on arrive sur un comportement récursif que l'on a déjà parcouru, on remplace les instances dangereuses

par ck et les autres par 0. Une instance de ck' est sûre si elle n'est pas contenue dans une récursion ou si la récursion contient une autre horloge (qui est nécessairement plus lente). On utilise des techniques classiques pour la gestion des comportements récursifs [Hue75]. On représente un comportement récursif par un graphe cyclique en mémoire et on prend garde à chaque parcours des comportements à ne pas boucler indéfiniment.

L'inférence simplifie les comportements au fur et à mesure du calcul, mais elle ne calcule pas forcément la représentation la plus simple d'un comportement. Cela n'est pas gênant puisque seule la réactivité des comportements nous intéresse. Par exemple, simplifier $\kappa + \kappa$ en κ peut être couteux dans le cas général, puisqu'il faut parcourir les deux comportements ou les mettre sous une forme normale. L'implémentation ne gère donc que les cas simples comme $0 + 0 \equiv 0$.

Analyse de réactivité en REACTIVEML classique

Les articles [MP13a, MP13b] présentent la même analyse de réactivité dans le cas de REACTIVEML classique, c'est-à-dire sans domaines réactifs. Elle a été implémentée dans le compilateur REACTIVEML à partir de la version 1.08.02. Elle se base sur le système de types de REACTIVEML et non le calcul d'horloges. Il n'y a donc pas d'horloge locale dans les jugements de typage ni d'horloge d'activation des processus. Puisqu'il n'y a qu'une seule horloge, on note $\bullet$ pour $\top_{ck}$ qui est le comportement associé à l'expression `pause`. Ce comportement vérifie des propriétés supplémentaires qui ne sont vraies que pour l'horloge globale. On a par exemple $\bullet + \kappa \equiv \kappa$ et $\bullet^\infty \equiv \bullet$. L'absence des domaines réactifs permet également de simplifier la preuve de sûreté, puisque l'on peut se passer par exemple des lemmes 7.8 et 7.9 mais aussi simplifier les définitions 9, 10 et 13 qui n'ont plus besoin de mentionner les horloges.

Depuis son implémentation dans le compilateur, l'analyse a été utilisée notamment pour l'écriture de l'interprète ANTESCOFO en REACTIVEML [BJMP13]. Ces expériences ont permis de montrer l'efficacité de l'analyse, qui atteint un bon compromis entre sa précision et la complexité de son implémentation. Le fait que l'on puisse étendre l'analyse au cadre des domaines réactifs montre bien la généralité de l'approche choisie.

Un des problèmes révélés par l'utilisation de cette analyse est qu'il faudrait ajouter au compilateur un moyen de ne pas afficher d'avertissement pour les processus dont on sait qu'il sont coopératifs mais que l'analyse ne sait pas traiter, typiquement parce qu'il s'agit d'un processus qui fait une récursion finie. En effet, dans le cas d'un long programme, le compilateur peut parfois afficher de nombreux avertissements dont on sait que l'on peut les ignorer. On peut alors rater parmi tous ces avertissements un autre avertissement qui révèle un vrai problème de réactivité, comme cela s'est passé pour l'interprète ANTESCOFO. L'autre difficulté est que les comportements calculés pour les processus récursifs sont difficiles à comprendre, comme par exemple pour le processus `fix_main` (page 128). Ce n'est pas très gênant puisque les comportements des processus ne sont pas faits pour être affichés et sont masqués par défaut.

Limites du système de types

Comme dans le cas du calcul d'horloges, le système de types-et-effets que l'on vient de décrire pose problème lorsque l'on utilise des processus comme des objets de première classe :

```
let process inst = ()
val inst : unit process['c0 | 0]

let process noinst = pause global_ck
val noinst : unit process['c0 | ?topck]

let l = [inst; noinst]
This expression has clock ((unit process['c0 | 0]) list),
but is used with clock ((unit process['c0 | ?topck]) list).
```

On ne peut pas stocker ensemble des processus qui ont des comportements différents, ce qui est le cas a priori de tous processus non triviaux. Dans le cas du calcul d'horloges, cela restreint l'expressivité du langage. Mais dans le cas de l'analyse de réactivité, on ne souhaite pas rejeter de programmes mais simplement afficher un message d'avertissement. Que faire alors si on ne peut pas donner un comportement à un processus ? Comment analyser le reste du programme ? Plutôt que de répondre à cette épineuse question, nous verrons dans le chapitre 8 que l'on peut étendre le système de types pour qu'il devienne une extension conservatrice du calcul d'horloges, c'est-à-dire qu'il soit capable de donner un comportement à n'importe quel programme correct du point de vue des types. On utilisera pour cela une forme de sous-typage restreint aux comportements que l'on appelle *subeffecting* [NNH99].

Autres modèles de concurrence

Nous pensons que cette analyse de réactivité peut s'appliquer dans d'autres modèles de concurrence coopérative. Il suffit alors de donner un comportement distinctif, noté par exemple •, aux opérations qui coopèrent avec l'ordonnanceur, telles que `yield` par exemple. On peut ensuite adapter les définitions pour vérifier qu'il n'existe pas de boucle infinie qui ne coopère jamais. Ainsi, on adapte la définition 9 pour définir un comportement non instantané comme un comportement dont toutes les alternatives contiennent le comportement •. On modifie ensuite la définition 10 de comportements réactifs pour qu'elle utilise la nouvelle définition de comportement non instantané. La distinction entre fonctions et processus est nécessaire pour ne pas afficher un message d'avertissement pour chaque définition de fonction récursive.

L'avantage du modèle synchrone est qu'il suffit que chaque processus coopère à chaque instant pour qu'un programme soit réactif, puisque les processus sont exécutés au même rythme. Dans le cas d'un autre modèle de concurrence, il faut faire des hypothèses sur l'équité de l'ordonnanceur pour arriver à la même conclusion. Ces hypothèses sont déjà faites dans la majorité des cas, comme par exemple pour CONCURRENT HASKELL [JGF96].

Travaux similaires

L'analyse des boucles instantanées et de la réactivité des programmes est un sujet ancien sur lequel beaucoup de travaux ont été menés, même récemment [AP13, AM13, Jef13]. Il est lié notamment au problème de la productivité des fonctions sur les listes [Sij89] ou encore aux conditions de gardes dans les assistants de preuves [BFG+04]. Nous ne cherchons donc pas ici à faire un état de l'art exhaustif du domaine et nous nous concentrons uniquement sur les travaux directement liés.

Notre analyse de réactivité se base sur le travail de [ANN99]. Il s'agit d'un système de types pour le langage CONCURENTML [Rep93], qui étend ML avec des primitives d'envoi de messages. Le comportement d'une fonction reflète de la même façon la structure du programme, mais les atomes du langage de comportements sont les émissions et réceptions sur les canaux de communication. Les auteurs ne présentent pas d'analyse statique générique, mais utilisent plutôt leur système de types sur des exemples particuliers. Par exemple, ils montrent dans le cas d'un programme implémentant un protocole simple que l'émission sur un premier canal précède toujours celle sur un second canal. L'idée d'utiliser un système de types pour prouver la réactivité ou de façon similaire la terminaison d'un programme n'est pas nouvelle. [Bou10] utilise cette approche pour prouver la terminaison d'un langage fonctionnel avec références, en stratifiant la mémoire pour éviter l'encodage de la récursion par des références.

L'analyse de réactivité est un sujet classique dans les langages synchrones. Dans le cas d'ESTEREL, la non réactivité d'un programme peut également être due au fait qu'il n'est pas causal, c'est-à-dire qu'il n'existe pas d'environnement de signaux cohérent permettant la réaction du programme. C'est typiquement le cas si un programme fait l'hypothèse qu'un signal est à la fois présent et absent :

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

Pour montrer la réactivité d'un programme, il faut donc d'abord montrer qu'il est causal, par exemple en utilisant la notion de causalité constructive [Ber96]. Le modèle de concurrence de REACTIVEC permet de garantir que tous les processus sont causaux « par construction ». Il suffit alors de vérifier que les boucles ne sont pas instantanées, ce que l'on appelle un programme *loop-safe* dans [Ber96]. Il est très simple de garantir cette condition en ESTEREL par une analyse presque syntaxique puisque le langage est du premier ordre et sans récursion [TdS05].

Plus proche de REACTIVEML, l'analyse de réactivité de FUNLOFT [AD07] ne garantit pas simplement que les instants terminent, mais donne aussi une borne sur la durée de chaque instant par une analyse de valeurs. Cette analyse se restreint à un langage du premier ordre. Dans le langage ULM [Bou04a], chaque appel récursif induit une pause implicite. Il est donc impossible d'avoir une récursion instantanée, mais au prix d'une perte d'expressivité. Par exemple, dans le cas du processus `server` (page 111), des messages pourraient être perdus puisqu'il se passerait un instant entre le lancement d'un processus reçu et l'attente d'une nouvelle réception.

CHAPITRE 8

Extensions des systèmes de types

Nous allons présenter dans ce chapitre plusieurs extensions du calcul d'horloges et de l'analyse de réactivité. La première est l'ajout de *polymorphisme de rang supérieur* dans le calcul d'horloges, qui rendra possible l'écriture de combinateurs utilisant des domaines réactifs. La seconde est une forme restreinte de sous-typage limitée aux effets appelée *subeffecting* [NNH99], que l'on implémente de façon nouvelle en s'inspirant des types rangées [Rém94]. Elle permet de donner un même type à deux processus ayant un effet ou un comportement différent. Cela permettra donc au calcul d'horloges d'accepter plus de programmes et à l'analyse de réactivité d'être une extension conservatrice du calcul d'horloges, c'est-à-dire de donner un comportement à tout programme bien typé.

8.1 Polymorphisme de rang supérieur dans le calcul d'horloges

Motivation

Le calcul d'horloges présenté dans le chapitre 6 répond à la contrainte de restriction de l'échappement de portée des signaux, mais il rejette beaucoup de programmes corrects. C'est en particulier le cas des combinateurs utilisant des domaines réactifs, comme le processus run_domain (page 108) :

exemples/ch3/run_domain.rml

```
let process run_domain q =
  domain ck do run q done
```

```
The clock of this expression, that is, '_a process {?ck|''_e0}
depends on ck which escapes its scope.
```

La dérivation de typage permet de mieux comprendre pourquoi ce processus est rejeté par le calcul d'horloges :

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma[x \mapsto \{\gamma\}], \gamma \vdash q : \beta \texttt{ process}\{\gamma|\phi\} \mid \emptyset}{\Gamma[x \mapsto \{\gamma\}], \gamma \vdash \texttt{run } q : \beta \mid \phi}}{\Gamma, \gamma' \vdash \texttt{domain } x \texttt{ by } \infty \texttt{ do } (\texttt{run } q) : \beta \mid \phi}}{\Gamma, \top_{ck} \vdash \texttt{process } (\texttt{domain } x \texttt{ by } \infty \texttt{ do } (\texttt{run } q)) : \beta \texttt{ process}\{\gamma'|\phi\} \mid \emptyset}}{\Gamma_0, \top_{ck} \vdash \lambda q.\texttt{process } (\texttt{domain } x \texttt{ by } \infty \texttt{ do } (\texttt{run } q)) : \beta \texttt{ process}\{\gamma|\phi\} \xrightarrow{\emptyset} \beta \texttt{ process}\{\gamma'|\phi\} \mid \emptyset}$$

où $\Gamma = \Gamma_0[q \mapsto \beta \texttt{ process}\{\gamma|\phi\}]$. Pour typer la fonction, on doit typer son corps dans l'environnement de typage étendu Γ. On introduit ensuite une variable de comportement γ' pour représenter l'horloge d'activation du processus. Le corps du processus est un domaine réactif, dont l'horloge locale est désignée par la variable γ. On finit la dérivation en lisant le type de q dans l'environnement, puis en appliquant la règle pour le lancement de processus. L'horloge locale γ du domaine s'échappe puisqu'elle apparait dans l'environnement dans le type de la variable q.

La source de ce problème est la règle de l'abstraction :

$$\frac{\Gamma \sqcup \{x \mapsto ct_2\}, ce \vdash e_1 : ct_1 \mid cf_1}{\Gamma, ce \vdash \lambda x.e_1 : ct_2 \xrightarrow{cf_1} ct_1 \mid \emptyset}$$

On remarque que l'on associe dans l'environnement le type ct_2 à l'argument de la fonction, et pas un schéma de type cs_2. En particulier, ce type ne peut contenir de variables d'horloge quantifiées universellement, ce qui devrait être le cas de l'horloge d'activation du processus q pour qu'on puisse le lancer à l'intérieur du domaine réactif. On a donc besoin de *polymorphisme de rang supérieur* [Rey74, Gir72], c'est-à-dire d'avoir une fonction dont le type de l'argument contient des variables quantifiées universellement. Ici, on souhaiterait que l'argument q ait le type $\forall\gamma.\ \beta\ \texttt{process}\{\gamma|\{\gamma\}\}$.

État de l'art

Le besoin de polymorphisme de rang supérieur est bien connu dès que l'on ajoute des types existentiels ou des contraintes d'échappement de portée dans ML[1] (voir [GR97] ou [JVWS07]). En ML [DM82], on distingue les types ct sans aucun quantificateur et les schémas de type cs qui n'apparaissent que dans l'environnement Γ. Si on permet l'utilisation libre de quantificateurs universels dans les types, alors on obtient SYSTEM F [Gir72, Rey74]. Le problème est que l'inférence de types devient alors indécidable [Wel99]. De nombreux travaux ont donc porté sur la recherche d'un compromis entre les possibilités offertes par SYSTEM F et la simplicité et décidabilité de ML.

Une première approche consiste à partir de SYSTEM F et tenter de réduire les annotations de types que le programmeur doit ajouter au programme en faisant de l'inférence locale [PT00]. L'autre approche consiste à partir de ML et ajouter des cas particuliers de polymorphisme de rang supérieur. On peut par exemple permettre l'utilisation de schémas de type pour les constructeurs [LO92] ou les champs d'enregistrements comme c'est le cas en OCAML. D'autres approches permettent d'annoter les arguments des fonctions [OL96] ou encore d'indiquer explicitement les instanciations mais sans avoir à donner les types [GR97]. Le polymorphisme de rang supérieur est intégré dans HASKELL [JVWS07] et combine l'approche de [OL96] avec l'inférence locale de [PT00] pour limiter au maximum les annotations de type. [Rém05] montre comment gérer le cas d'un langage avec effets de bords.

Toutes ces méthodes sont *prédicatives*, c'est-à-dire que les variables de type sont toujours instanciées par des types monomorphes, jamais par des schémas de type. On peut passer outre cette limitation mais au prix d'une grande complexité dans la théorie et l'implémentation, qui s'éloigne de ML. La solution la plus prometteuse est sans doute MLF [LBR09] qui ajoute l'idée de *quantification flexible*, c'est-à-dire de quantification sur l'ensemble des instances d'un schéma de type. MLF représente un certain optimal en termes de minimalité des annotations, mais autant sa théorie que son implémentation sont très complexes. [VWPJ08] et [Lei09] s'inspirent de cette approche mais avec d'autres compromis entre simplicité et expressivité.

Implémentation

Nous avons choisi d'implémenter une version élémentaire de polymorphisme de rang supérieur. La première possibilité est de déclarer comme en OCAML un champ d'enregistrement polymorphe :

exemples/ch8/run_domain_record.rml

```
type poly_process =
  { l : 'ck. unit process{'ck|'ck} }

let process run_domain q =
  domain ck do run q.l done
val run_domain : poly_process =>{0} unit process {'c0|0}
```

1. On peut par exemple remarquer que l'article [LO92] dont s'inspire le calcul d'horloges est suivi d'un article sur le polymorphisme de rang supérieur [OL96].

```
let process main =
  let process q = pause in
  run run_domain { l = q }
val main : unit process {'c0|0}
```

Le type `poly_process` est le type des processus dont l'horloge d'activation `'ck` est quantifiée universellement. On obtient un programme bien typé puisque la dérivation de typage est maintenant :

$$\dfrac{\dfrac{\dfrac{\Gamma[x \mapsto \{\gamma\}], \gamma \vdash q.l : \texttt{unit process}\{\gamma|\{\gamma\}\} \mid \emptyset}{\Gamma[x \mapsto \{\gamma\}], \gamma \vdash \texttt{run}\ q.l : \texttt{unit} \mid \{\gamma\}} \quad \gamma \notin ftv(\Gamma, \texttt{unit})}{\Gamma, \gamma' \vdash \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ (\texttt{run}\ q.l) : \texttt{unit} \mid \emptyset}}{\Gamma, \top_{ck} \vdash \texttt{process}\ (\texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ (\texttt{run}\ q.l)) : \texttt{unit process}\{\gamma'|\phi\} \mid \emptyset}}{\Gamma_0, \top_{ck} \vdash \lambda q.\texttt{process}\ (\texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ (\texttt{run}\ q.l)) : \texttt{poly\_process} \xrightarrow{\emptyset} \texttt{unit process}\{\gamma'|\emptyset\} \mid \emptyset}$$

où $\Gamma = \Gamma_0 \sqcup \{q \mapsto \texttt{poly\_process}\}$. L'horloge locale γ du domaine n'apparait plus dans le type de $q.l$, puisque celui-ci est un schéma de type dans lequel l'horloge d'activation du processus est quantifiée universellement. On obtient donc une fonction qui prend en argument une valeur de type `poly_process` et renvoie un processus d'horloge d'activation γ' non contrainte et d'effet nul.

Devoir déclarer un type enregistrement pour chaque combinateur est très lourd. Il est donc possible de simplement ajouter une annotation sur l'argument de la fonction :

exemples/ch8/run_domain_annot.rml

```
let process run_domain
    (q : 'ck. 'a process{'ck|'ck}) =
  domain ck do run q done
val run_domain : (forall 'c0. 'a process {'c0|'c0}) =>{0} 'a process {'c1|0}

let process main =
  let process q = pause in
  run run_domain q
val main : unit process {'c0|0}
```

On a alors formellement deux règles pour l'abstraction, selon la présence ou non d'une annotation :

$$(\textsc{Abs})\ \dfrac{\Gamma \sqcup \{x \mapsto ct_2\}, ce \vdash e_1 : ct_1 \mid cf_1}{\Gamma, ce \vdash \lambda x.e_1 : ct_2 \xrightarrow{cf_1} ct_1 \mid \emptyset} \qquad (\textsc{AbsAnnot})\ \dfrac{\Gamma \sqcup \{x \mapsto cs_2\}, ce \vdash e_1 : ct_1 \mid cf_1}{\Gamma, ce \vdash \lambda(x : cs_2).e_1 : cs_2 \xrightarrow{cf_1} ct_1 \mid \emptyset}$$

On infère donc uniquement des types ct, jamais de schémas de type cs. L'inférence reste donc décidable et les modifications sur l'algorithme d'inférence sont minimes. La contrepartie est que notre approche est très fragile. Par exemple, le processus suivant dans lequel on a simplement appliqué l'identité sur l'argument `q` est rejeté par le calcul d'horloges :

exemples/ch9/run_domain_annot_wrong.rml

```
let process run_domain_wrong
    (q : 'ck. 'a process{'ck|'ck}) =
  let q = (fun x -> x) q in
  domain ck do run q done
The clock of this expression, that is, '_a process {?ck|''_e0}
depends on ck which escapes its scope.
```

En effet, le type de `q` n'est pas généralisé puisqu'il s'agit du résultat d'une application qui est une expression expansive dont le type n'est pas généralisable.

Cette implémentation élémentaire de polymorphisme de rang supérieur nous semble suffisante pour la grande majorité des usages. En effet, on n'a besoin d'annotations que si on utilise des processus d'ordre supérieur dans un domaine, ce qui est un cas finalement peu courant.

Remarque 9. Le type `'ck. 'a process{'ck|'ck}` des processus que l'on a mis dans l'annotation impose au processus d'avoir un effet réduit à son horloge d'activation. Notre algorithme d'inférence ne sait pas traiter les cas plus complexes, comme le fait d'avoir un effet sur deux horloges (`'ck. 'a process{'ck|'ck + ?topck}`) ou d'avoir un effet au moins sur l'horloge locale (`'ck. 'a process{'ck|'ck + 'f}`). En effet, cela requiert d'implémenter l'unification des ensembles d'effets, c'est-à-dire l'unification modulo associativité, commutativité et idempotence (ACI). Celle-ci est décidable [DPR01] mais avec une grande complexité. L'extension du calcul d'horloges pour gérer ces cas plus complexes constitue une piste de recherche intéressante.

8.2 Analyse de réactivité avec rangées

Présentation

L'analyse de réactivité du chapitre 7 pose également des problèmes lorsque l'on utilise des processus comme valeurs de première classe. Ainsi, le programme suivant est rejeté :

```
let process inst = ()
val inst : unit process['c0 | 0]

let process noinst = pause global_ck
val noinst : unit process['c0 | ?topck]

let l = [inst; noinst]
This expression has clock ((unit process['c0 | 0]) list),
but is used with clock ((unit process['c0 | ?topck]) list).
```

En effet, les valeurs dans la liste `l` doivent avoir la même type, et donc les processus doivent avoir exactement le même comportement.

Pour résoudre ce problème, nous allons introduire dans le système de types un cas particulier de sous-typage appliqué aux comportements, ce que l'on appelle du *subeffecting* [NNH99]. L'idée est que le comportement d'un processus n'est plus *égal* au comportement de son corps, mais est *au moins* le comportement de son corps. On peut toujours en effet remplacer un comportement par un autre plus grand, c'est-à-dire moins précis, sans que cela ne change la sûreté du système de types. La règle PROCESS de typage des définitions de processus devient :

$$(\text{PROCESS'})\frac{\Gamma, ce' \vdash e_1 : rt \mid \kappa_1}{\Gamma, ce \vdash \texttt{process}\ e_1 : rt\ \texttt{process}\{ce' | \kappa_1 + \kappa'\} \mid 0}$$

Cette modification s'inspire également des *types rangées* [Rém94, Wan87] qui, à l'origine, permettent de gérer un autre cas particulier de sous-typage : les enregistrements extensibles et les objets. L'idée est d'ajouter dans le type d'un enregistrement une variable libre, que l'on appellera *variable de rangée*, pour représenter tous les autres champs de l'enregistrement. Dans notre cas, la variable de rangée représente tous les autres comportements possibles du processus. Ainsi, le type inféré pour un processus contient toujours cette variable de rangée. L'exemple précédent devient alors :

```
let process inst = ()
val inst : unit process['c0 | 0 + 'r0]

let process noinst = pause global_ck
val noinst : unit process['c0 | ?topck + 'r0]

let l = [inst; noinst]
val l : (unit process['c0 | ?topck + 0 + 'r0]) list
```

Grâce à ces variables de rangées, on a pu donner un comportement aux processus de la liste, qui est soit celui du premier processus, soit celui du deuxième, soit encore un autre grâce à la

variable de rangée. Le problème de cette approche est qu'elle crée un grand nombre de variables de comportement, comme le montre l'exemple suivant :

```
let process p = pause
val p : unit process['c0 | {'c0} + 'r0]
let process q = run p
val q : unit process['c0 | run ({'c0}+ 'r0) + 'r1]
let process r = run q
val r : unit process['c0 | run (run ({'c0} + 'r0) + 'r1) + 'r2]
```

La solution à ce problème est d'utiliser ce que l'on appelle le *masquage d'effets* [LG88] :

$$(\textsc{Mask})\ \frac{\Gamma, ce \vdash e : rt \mid \kappa \qquad \phi \in fbv(\kappa) \setminus fbv(\Gamma, rt)}{\Gamma, ce \vdash e : rt \mid \kappa[\phi \leftarrow \bullet]}$$

L'intuition de cette règle est que si la variable de rangée ϕ est libre dans l'environnement de typage, alors elle n'est pas contrainte puisque l'on n'unifie jamais les comportements en profondeur comme nous le verrons plus tard. On peut donc la remplacer par n'importe quelle valeur. On choisit ici le comportement $\bullet$ que l'on ajoute dans le langage de comportements et qui est l'élément neutre de l'opérateur $+$. On obtient alors des comportements bien plus petits dans notre exemple :

```
let process p = pause
val p : unit process['c0 | {'c0} + 'r0]
let process q = run p
val q : unit process['c0 | run {'c0} + 'r0]
let process r = run q
val r : unit process['c0 | run (run ({'c0})) + 'r0]
```

Remarque 10. La preuve de sûreté de l'analyse, présentée dans la partie 7.4, est toujours valable avec ces nouvelles règles. Pour le cas de la définition d'un processus, on utilise uniquement le fait que le comportement du corps est plus petit, ce qui est toujours le cas. On peut utiliser l'approche de [CHT02] pour traiter le cas du masquage d'effets. L'intuition de la preuve est que, puisque la variable ϕ n'apparait ni dans l'environnement ni dans le type de retour, on peut la substituer par le comportement $\bullet$ et garder un terme bien typé. On procède ainsi de le même manière que pour la règle DOMAIN dans la preuve de sûreté du calcul d'horloges.

L'ajout des rangées dans le système de types permet également de gérer le cas des références. Elles permettent en effet d'encoder la récursion, comme dans l'exemple suivant qui crée un processus qui boucle instantanément :

exemples/ch8/landin.rml

```
let landin () =
  let f = ref (process ()) in
  f := process (print_endline "Coucou"; run !f);
  !f
val landin : unit -> unit process['c0 | 0 + (rec 'r1. run (0 + 'r1)) + 'r2]
Warning: This expression may produce an instantaneous recursion.
```

Après la création de la référence, on obtient le comportement $\kappa_1 = 0 + \phi_1$ pour f. Lorsque l'on type l'affectation, on doit unifier ce comportement avec celui du processus anonyme, c'est-à-dire run $\kappa_1 + \phi_1'$. Pour cela, on choisit $\phi_1 \leftarrow$ run $\kappa_1 + \phi_2$ et $\phi_1' \leftarrow 0 + \phi_2$ où ϕ_2 est une variable fraiche. On obtient alors une rangée récursive $0 + (\mu\phi_1.\,\texttt{run}\ (0 + \phi_1)) + \phi_2$. Nous décrirons plus précisément l'algorithme d'unification dans la partie suivante. Puisque notre analyse ne contient pas de cas particulier pour les processus récursifs et se base uniquement sur l'unification, elle permet donc de détecter cet autre cas de récursion instantanée. L'ajout des rangées permet de stocker des processus avec des comportements différents dans la même référence.

Implémentation

L'utilisation de variables de rangées pour le *subeffecting* permet d'implémenter le sous-typage uniquement par unification, sans utiliser d'ensemble de contraintes comme dans les approches habituelles [TJ92, ANN99]. Cela simplifie son intégration dans n'importe quelle implémentation d'inférence de ML et en particulier dans celle du compilateur REACTIVEML. Il suffit d'utiliser l'algorithme d'unification de rangées de comportements que nous allons décrire dans la suite.

Au cours de l'unification, le comportement d'un processus est toujours soit une variable de comportement ϕ, soit une rangée $\kappa + \phi$, soit une rangée récursive $\mu\phi'.\,(\kappa + \phi)$ avec $\phi' \in fbv(\kappa)$. On aurait pu mettre en avant cet invariant en modifiant la syntaxe des comportements pour distinguer les comportements κ et les rangées de comportements ρ :

$$\begin{array}{ll}
\kappa ::= 0 \mid ce \mid \phi \mid \kappa \mid\mid \kappa \mid \kappa + \kappa \mid \kappa; \kappa \mid \mu\phi.\,\kappa \mid \texttt{run}\ \rho \mid \texttt{domain}\ \kappa & \text{(comportements)} \\
\rho ::= \kappa + \rho \mid \varrho \mid \mu\varrho.\,\rho & \text{(rangées de comportements)} \\
rt ::= \cdots \mid rt\ \texttt{process}\{ce|\rho\} & \text{(types)}
\end{array}$$

Nous avons choisi ici de rester avec la syntaxe la plus simple, que nous avons décrite au début de la partie 7.3. Nous verrons l'autre approche dans le cas du calcul d'horloges avec rangées dans la partie 8.3.

Algorithme d'inférence Nous allons maintenant décrire plus formellement l'unification en présence de variables de rangées. Nous commençons dans un premier temps par une version de l'algorithme $\mathcal{W}$ [DM82] traditionnellement utilisé pour l'inférence de types en ML, que l'on étend ici avec le calcul des comportements. L'algorithme prend en entrée un environnement de typage Γ, une horloge locale ce et une expression e et renvoie une substitution θ, un type rt et un comportement κ :

$$\mathcal{W}(\Gamma, ce, e) = \theta, rt, \kappa$$

L'algorithme $\mathcal{W}$ utilise pour unifier les types l'algorithme $\mathcal{U}$ que nous décrirons un peu plus loin. On note id la substitution vide, c'est-à-dire l'identité, θrt l'application de la substitution θ au type rt et $\theta_2\theta_1$ la composition des substitutions θ_1 et θ_2, où on applique d'abord θ_1 puis θ_2. La figure 8.1 montre la définition de cet algorithme sur une partie des expressions du langage. Les constructions restantes peuvent se traiter de la même façon. Il s'agit de l'algorithme classique dans lequel on a ajouté le calcul des comportements, qui ne requiert aucune unification :

- L'expression $()$ a le type `unit`, le comportement 0 et renvoie la substitution id puisqu'aucune unification n'est nécessaire.
- Pour obtenir le type d'une variable, on instancie son schéma de type $\forall\bar{\alpha}.\forall\bar{\phi}.\,rt$ par des variables de type $\bar{\beta}$ et de comportement $\bar{\psi}$ fraiches.
- Lors du typage d'une fonction, on type son corps e_1 en utilisant une variable fraiche β pour le type de l'argument x. On retourne ensuite un type fonction associant à cette variable fraiche le type rt_1 du corps de la fonction.
- Pour l'application, on calcule d'abord le type et le comportement des deux sous-expressions e_1 et e_2. On unifie ensuite le type de e_1 avec le type $rt_2 \to \beta$ pour s'assurer qu'il s'agit bien d'une fonction prenant en argument des valeurs de type rt_2. On retourne enfin le type β de retour de la fonction, auquel on doit appliquer la substitution θ_3 résultant de l'unification.
- Dans le cas d'une récursion, on associe à la variable x une variable fraiche β pour typer le corps. On unifie ensuite β avec le type rt_1 calculé pour le corps.
- Pour la construction `let/and/in`, on calcule le type et le comportement des deux branches. On généralise ensuite leurs types avant de les ajouter à l'environnement de typage Γ' utilisé pour typer le corps e_3.

Les deux règles intéressantes sont celles qui concernent les processus :

- Lors de la création d'un processus, on calcule d'abord le type rt_1 et le comportement κ_1 du corps e_1 en utilisant une variable fraiche γ pour l'horloge locale. Le type du processus est alors donné par le type du corps, cette horloge locale et le comportement du corps auquel on ajoute une variable de rangée ϕ fraiche.

$$
\begin{aligned}
\mathcal{W}(\Gamma, ce, ()) &= (id, \texttt{unit}, 0) \\
\mathcal{W}(\Gamma, ce, x) &= (id, rt[\bar{\alpha} \leftarrow \bar{\beta}; \bar{\phi} \leftarrow \bar{\psi}], 0) \text{ où } \Gamma(x) = \forall \bar{\alpha}. \forall \bar{\phi}.\, rt && \bar{\beta}, \bar{\psi} \text{ frais} \\
\mathcal{W}(\Gamma, ce, \lambda x.e_1) &= \texttt{let}\ (\theta_1, rt_1, \kappa_1) = \mathcal{W}(\Gamma \sqcup \{x \mapsto \beta\}, ce, e_1) && \beta \text{ frais} \\
&\quad \texttt{in}\ (\theta_1, \theta_1\beta \to rt_1, 0) \\
\mathcal{W}(\Gamma, e_1\ e_2) &= \texttt{let}\ (\theta_1, rt_1, \kappa_1) = \mathcal{W}(\Gamma, ce, e_1) \\
&\quad \texttt{let}\ (\theta_2, rt_2, \kappa_2) = \mathcal{W}(\theta_1\Gamma, \theta_1 ce, e_2) \\
&\quad \texttt{let}\ \theta_3 = \mathcal{U}(\theta_2 rt_1, rt_2 \to \beta) && \beta \text{ frais} \\
&\quad \texttt{in}\ (\theta_3\theta_2\theta_1, \theta_3\beta, 0) \\
\mathcal{W}(\Gamma, \texttt{rec}\ x = e_1) &= \texttt{let}\ (\theta_1, rt_1, \kappa_1) = \mathcal{W}(\Gamma \sqcup \{x \mapsto \beta\}, ce, e_1) && \beta \text{ frais} \\
&\quad \texttt{let}\ \theta_2 = \mathcal{U}(\theta_1\beta, rt_1) \\
&\quad \texttt{in}\ (\theta_2\theta_1, \theta_2 rt_1, 0) \\
\mathcal{W}(\Gamma, \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3) &= \texttt{let}\ (\theta_1, rt_1, \kappa_1) = \mathcal{W}(\Gamma, ce, e_1) \\
&\quad \texttt{let}\ (\theta_2, rt_2, \kappa_2) = \mathcal{W}(\theta_1\Gamma, \theta_1 ce, e_2) \\
&\quad \texttt{let}\ \Gamma' = \theta_2\theta_1\Gamma \sqcup \{x_1 \mapsto gen(\theta_2 rt_1, e_1, \theta_2\theta_1\Gamma); \\
&\qquad\qquad x_2 \mapsto gen(rt_2, e_2, \theta_2\theta_1\Gamma)\} \\
&\quad \texttt{let}\ (\theta_3, rt_3, \kappa_3) = \mathcal{W}(\Gamma', \theta_2\theta_1 ce, e_3) \\
&\quad \texttt{in}\ (\theta_3\theta_2\theta_1, rt_3, (\theta_3\theta_2\kappa_1 \mid\mid \theta_3\kappa_2); \kappa_3) \\
\mathcal{W}(\Gamma, ce, \texttt{process}\ e_1) &= \texttt{let}\ (\theta_1, rt_1, \kappa_1) = \mathcal{W}(\Gamma, \gamma, e_1) && \gamma \text{ frais} \\
&\quad \texttt{in}\ (\theta_1, rt_1\ \texttt{process}\{\theta_1\gamma | \kappa_1 + \phi\}, 0) && \phi \text{ frais} \\
\mathcal{W}(\Gamma, ce, \texttt{run}\ e_p) &= \texttt{let}\ (\theta_p, rt_p, \kappa_p) = \mathcal{W}(\Gamma, ce, e_p) \\
&\quad \texttt{let}\ \theta_2 = \mathcal{U}(rt_p, \beta\ \texttt{process}\{\theta_p ce | \phi\}) && \beta, \phi \text{ frais} \\
&\quad \texttt{in}\ (\theta_2\theta_p, \theta_2\beta, \texttt{run}\ \theta_2\phi)
\end{aligned}
$$

FIGURE 8.1 – Algorithme $\mathcal{W}$ avec calcul des comportements

– Lors du lancement d'un processus, on s'assure que e_1 est un processus sur l'horloge locale ce en unifiant son type rt_p avec un processus dont le type de retour est une variable fraiche β et le comportement est une variable de comportement fraiche ϕ.

Algorithme d'unification L'algorithme $\mathcal{U}$ d'unification des types et comportements est défini sur la figure 8.2. Il prend en entrée deux types et renvoie une substitution θ qui rend les deux types égaux ou échoue avec la valeur spéciale `fail` :

$$\mathcal{U}(rt_1, rt_2) = \theta$$

L'unification des types est définie sur la figure 8.2a et correspond à la définition habituelle. On unifie uniquement les types de même forme, composante par composante. Dans le cas d'une variable, on vérifie avec la fonction `occur_check` si la variable apparait dans le corps du type. Si c'est le cas, on échoue pour ne pas créer de type récursif. Sinon, on assigne une nouvelle valeur à la variable. Toute la nouveauté se situe dans l'unification des comportements, définie sur la figure 8.2b :

– La deuxième règle énonce l'unification classique des types récursifs [Hue75]. On crée un comportement récursif si on unifie une variable avec un comportement contenant cette variable.
– La suivante montre l'unification des rangées. On créé une variable de rangée ϕ fraiche, puis on donne les bonnes valeurs aux deux variables de rangées ϕ_1 et ϕ_2 pour obtenir des comportements égaux.

$$
\begin{aligned}
\mathcal{U}(T_1, T_1) &= id \\
\mathcal{U}(rt_1 \times rt_2, rt'_1 \times rt'_2) &= \texttt{let}\ \theta_1 = \mathcal{U}(rt_1, rt'_1) \\
&\quad\ \texttt{let}\ \theta_2 = \mathcal{U}(\theta_1 rt_2, \theta_1 rt'_2) \\
&\quad\ \texttt{in}\ \theta_2\theta_1 \\
\mathcal{U}(rt_1 \to rt_2, rt'_1 \to rt'_2) &= \texttt{let}\ \theta_1 = \mathcal{U}(rt_1, rt'_1) \\
&\quad\ \texttt{let}\ \theta_2 = \mathcal{U}(\theta_1 rt_2, \theta_1 rt'_2) \\
&\quad\ \texttt{in}\ \theta_2\theta_1 \\
\mathcal{U}(rt\ \texttt{process}\{ce|\kappa\}, rt'\ \texttt{process}\{ce'|\kappa'\}) &= \texttt{let}\ \theta_1 = \mathcal{U}(rt, rt') \\
&\quad\ \texttt{let}\ \theta_2 = \mathcal{U}_{ce}(\theta_1 ce, \theta_1 ce') \\
&\quad\ \texttt{let}\ \theta_3 = \mathcal{U}_{\kappa}(\theta_2\theta_1\kappa, \theta_2\theta_1\kappa') \\
&\quad\ \texttt{in}\ \theta_3\theta_2\theta_1 \\
\mathcal{U}((rt_1, rt_2)\texttt{event}\{ce\}, (rt'_1, rt'_2)\texttt{event}\{ce'\}) &= \texttt{let}\ \theta_1 = \mathcal{U}(rt_1, rt'_1) \\
&\quad\ \texttt{let}\ \theta_2 = \mathcal{U}(\theta_1 rt_2, \theta_1 rt'_2) \\
&\quad\ \texttt{let}\ \theta_3 = \mathcal{U}_{ce}(\theta_2\theta_1 ce, \theta_2\theta_1 ce') \\
&\quad\ \texttt{in}\ \theta_3\theta_2\theta_1 \\
\mathcal{U}(rt, \alpha) = \mathcal{U}(\alpha, rt) &= \begin{cases} \texttt{fail} & \text{si } \texttt{occur\_check}(\alpha, rt) \\ [\alpha \leftarrow rt] & \text{sinon} \end{cases} \\
\mathcal{U}(rt_1, rt_2) &= \texttt{fail} \qquad \text{dans tous les autres cas}
\end{aligned}
$$

(a) Unification des types

$$
\begin{aligned}
\mathcal{U}_{\kappa}(\kappa, \kappa) &= id \\
\mathcal{U}_{\kappa}(\phi, \kappa) = \mathcal{U}_{\kappa}(\kappa, \phi) &= \begin{cases} [\phi \leftarrow \mu\phi.\,\kappa] & \text{si } \texttt{occur\_check}(\phi, \kappa) \\ [\phi \leftarrow \kappa] \end{cases} \\
\mathcal{U}_{\kappa}(\kappa_1 + \phi_1, \kappa_2 + \phi_2) &= [\phi_1 \leftarrow \kappa_2 + \phi; \phi_1 \leftarrow \kappa_1 + \phi] \qquad \phi \text{ frais} \\
\mathcal{U}_{\kappa}(\mu\phi'_1.\,(\kappa_1 + \phi_1), \kappa_2) = \mathcal{U}_{\kappa}(\kappa_2, \mu\phi'_1.\,(\kappa_1 + \phi_1)) &= \texttt{let}\ K_1 = \mu\phi'_1.\,(\kappa_1 + \phi_1) \\
&\quad\ \texttt{in}\ \mathcal{U}_{\kappa}(\kappa_1[\phi'_1 \leftarrow K_1] + \phi_1, \kappa_2)
\end{aligned}
$$

(b) Unification des comportements

FIGURE 8.2 – Unification des types et comportements

– Enfin, la dernière règle exprime l'unification de rangées récursives. Il faut déplier le comportement récursif pour ensuite appliquer l'unification de rangées. Dans l'implémentation où l'on utilise des graphes récursifs pour représenter les comportements récursifs, on unifie simplement le corps des comportements récursifs.

On peut remarquer que l'unification de comportements n'échoue jamais. L'analyse de réactivité avec rangées est donc une extension conservatrice du calcul d'horloges : on peut donner un comportement à tout programme bien typé. L'autre particularité de cette unification est qu'elle ne se fait que sur la tête des comportements, jamais en profondeur. Cela explique pourquoi le masquage d'effets est possible.

Travaux similaires

Le *subeffecting* [TJ92, NNH99] est traditionnellement exprimé sous la forme d'une règle non dirigée par la syntaxe. On reprend la syntaxe de l'analyse de réactivité pour simplifier la comparaison :

$$\frac{\Gamma, ce \vdash e : rt \mid \kappa \qquad \kappa \sqsubseteq \kappa'}{\Gamma, ce \vdash e : rt \mid \kappa'} \tag{8.1}$$

où $\sqsubseteq$ est une relation de sous-typage sur les effets. Dans le cas où les effets sont des ensembles (de régions par exemple), il s'agit de l'inclusion.

Une approche similaire est utilisée dans [ANN99]. L'idée est de forcer syntaxiquement les comportements à être *simples*, c'est-à-dire que les comportements sur les flèches ne peuvent être que des variables. On ajoute un ensemble de contraintes C au système de types pour conserver les relations entre les variables, que l'on exprime avec la relation $\sqsubseteq$. On exprime alors le subeffecting avec une règle de la forme :

$$\frac{\Gamma, C, ce' \vdash e_1 : rt \mid \kappa_1}{\Gamma, C \cup \{\kappa_1 \sqsubseteq \phi\}, ce \vdash \texttt{process}\ e_1 : rt\ \texttt{process}\{ce'|\phi\} \mid 0} \tag{8.2}$$

Nous pensons que les formulations 8.1 et 8.2 et notre approche du subeffecting sont équivalentes. En effet, notre formulation et celle de 8.2 sont des versions dirigées par la syntaxe de 8.1, où l'on applique le subeffecting uniquement au niveau des processus et des fonctions. Dans le cas des comportements, la relation de sous-typage est définie par :

$$\kappa_1 \sqsubseteq \kappa_1 + \kappa_2 \qquad \kappa_2 \sqsubseteq \kappa_1 + \kappa_2 \qquad \frac{\kappa_1 \equiv \kappa_2}{\kappa_1 \sqsubseteq \kappa_2}$$

Notre utilisation de variables de rangées pour le subeffecting est similaire à l'approche de [LP00] qui est une analyse des exceptions potentiellement levées par une fonction en OCAML. L'analyse de causalité de LUCID SYNCHRONE [CP01] s'appuie sur les mêmes principes. La différence avec notre approche est que cette analyse d'exceptions est une application plus directe des types rangées [Rém94], alors que notre cas est différent puisque les rangées ne contiennent pas d'étiquettes. Enfin, les travaux sur les objets en OCAML [RV97], également dérivés des types rangées, sont une source d'inspiration de ce travail (notamment de la distinction entre rangée ouverte et rangée fermée que l'on verra dans la prochaine partie). Ils montrent plusieurs évolutions du langage nécessaires lorsque l'on ajoute une dose de sous-typage basée sur les variables de rangées, comme par exemple les abréviations de type.

8.3 Calcul d'horloges avec rangées

Nous allons maintenant utiliser la même approche de *subeffecting* dans le cas du calcul d'horloges. Nous allons donc distinguer les effets et les rangées d'effets. Nous allons également utiliser la même notion de rangées pour les horloges des signaux. Cela permet de stocker ensemble des signaux avec des horloges différentes. Du point de vue technique, nous allons distinguer les rangées ouvertes et fermées, à la manière des objets en OCAML [RV97]. Le calcul d'horloges avec rangées accepte en particulier tout programme REACTIVEML bien typé n'utilisant pas de domaines réactifs.

Types

On change la syntaxe des types (définie page 96) pour ajouter les rangées d'horloges et d'effets :

$$
\begin{array}{lll}
ct ::= T \mid \alpha \mid \{cr\} \mid ct \times ct \mid (ct, ct)\texttt{event}\{|cr|\} & & \text{(types)} \\
\quad \mid ct \xrightarrow{fr} ct \mid ct\ \texttt{process}\{ce||fr\} & & \\
ce ::= \gamma \mid ck & & \text{(horloges)} \\
cr ::= \rho \mid ce + cr \mid \star & & \text{(rangées d'horloges)} \\
cf ::= \emptyset \mid \{cr\} \mid fr \mid cf \cup cf & & \text{(effets)} \\
fr ::= \psi \mid \star \mid cf + fr \mid \mu\psi.\, fr & & \text{(rangées d'effets)} \\
cs ::= ct \mid \forall\alpha.\, cs \mid \forall\gamma.\, cs \mid \forall\rho.\, cs \mid \forall\psi.\, cs & & \text{(schémas de type)} \\
\Gamma ::= \{x_1 \mapsto cs_1; \ldots; x_p \mapsto cs_p\} & & \text{(environnements)}
\end{array}
$$

Les signaux $(ct, ct)\texttt{event}\{|cr|\}$ et les types singletons $\{cr\}$ sont paramétrés par des rangées d'horloges cr. Ce sont des listes d'horloges terminées par une variable de rangée d'horloge ρ ou la rangée vide $\star$. On dit qu'une rangée est ouverte si elle se termine par une variable de rangée et qu'elle est fermée si elle se termine par la rangée vide. On associe maintenant aux fonctions $ct \xrightarrow{fr} ct$ et processus $ct\ \texttt{process}\{ce||fr\}$ une rangée d'effets fr. Les rangées d'effets fr reprennent la même structure que les rangées d'horloges, auxquelles on ajoute les rangées récursives $\mu\psi.\, fr$. On peut remarquer que les effets cf et rangées d'effets fr sont définis récursivement, puisqu'un effet peut contenir une rangée d'effets. En outre, il n'y a plus de variables d'effet ϕ puisque les effets n'apparaissent plus directement dans les types. Les rangées d'horloges et d'effets sont commutatives et les rangées récursives vérifient la propriété habituelle :

$$\mu\psi.\, fr = fr[\psi \leftarrow \mu\psi.\, fr] \qquad ce_1 + (ce_2 + cr) = ce_2 + (ce_1 + cr) \qquad cf_1 + (cf_2 + fr) = cf_2 + (cf_1 + fr)$$

Enfin, on définit une opération de généralisation partielle qui ne généralise que la variable de rangée d'horloges d'un signal :

$$
\begin{array}{ll}
gen_{cr}((ct_1, ct_2)\texttt{event}\{|cr|\}, e, \Gamma) = (ct_1, ct_2)\ \texttt{event}\{cr|\} & \text{si } e \text{ est expansive} \\
gen_{cr}((ct_1, ct_2)\texttt{event}\{|cr|\}, e, \Gamma) = \forall\bar{\rho}.\ (ct_1, ct_2)\ \texttt{event}\{cr|\} & \text{sinon, avec } \bar{\rho} = fcrv(cr) \setminus fcrv(\Gamma)
\end{array}
$$

où $fcrv(cr)$ désigne les variables de rangées d'horloges libres dans cr.

Règles

Les jugements de typage ont toujours la même forme :

$$\Gamma, ce \vdash e : ct \mid cf$$

qui signifie que dans l'environnement de typage Γ et l'horloge locale ce, l'expression e a le type ct et l'effet cf. L'environnement de typage initial Γ_0 est maintenant :

$$
\begin{array}{l}
\Gamma_0 \triangleq [\, global\_ck : \forall\rho.\, \{\top_{ck} + \rho\};\ \texttt{pause} : \forall\rho, \psi.\, \{\rho\} \xrightarrow{\{\rho\}+\psi} \texttt{unit};\ \texttt{qpause} : \forall\rho, \psi.\, \{\rho\} \xrightarrow{\{\rho\}+\psi} \texttt{unit}; \\
\qquad \texttt{last} : \forall\alpha_1, \alpha_2, \rho, \psi.\, (\alpha_1, \alpha_2)\texttt{event}\{|\rho|\} \xrightarrow{\{\rho\}+\psi} \alpha_2; \\
\qquad \texttt{emit} : \forall\alpha_1, \alpha_2, \rho, \psi_1, \psi_2.\, (\alpha_1, \alpha_2)\texttt{event}\{|\rho|\} \xrightarrow{\emptyset+\psi_1} \alpha_1 \xrightarrow{\{\rho\}+\psi_2} \texttt{unit}; \\
\qquad \texttt{true} : \texttt{bool};\ \texttt{fst} : \forall\alpha_1, \alpha_2, \psi.\, \alpha_1 \times \alpha_2 \xrightarrow{\emptyset+\psi} \alpha_1; \ldots]
\end{array}
$$

Le type de l'horloge globale $global\_ck$ est maintenant un type singleton associé à une rangée ouverte contenant l'horloge globale $\top_{ck}$. Sa variable de rangée ρ est quantifiée universellement pour que les différentes utilisations de la variable puissent avoir des types différents. On a ajouté des variables de rangées pour les effets de toutes les fonctions. Par exemple, l'opération `pause` est caractérisée par

$$\frac{\Gamma[x \mapsto ct_2], ce \vdash e_1 : ct_1 \mid cf_1}{\Gamma, ce \vdash \lambda x.e_1 : ct_2 \xrightarrow{cf_1+fr} ct_1 \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_1 : ct_2 \xrightarrow{fr} ct_1 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_2 \mid \emptyset}{\Gamma, ce \vdash e_1\ e_2 : ct_1 \mid fr}$$

$$\frac{\Gamma, ce' \vdash e_1 : ct \mid cf_1}{\Gamma, ce \vdash \texttt{process}\ e_1 : ct\ \texttt{process}\{ce' || cf_1 + fr\} \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_p : ct\ \texttt{process}\{ce||fr\} \mid \emptyset}{\Gamma, ce \vdash \texttt{run}\ e_p : ct \mid fr}$$

$$\frac{}{\Gamma, ce \vdash \texttt{local\_ck} : \{ce + cr\} \mid \emptyset}$$

$$\text{(SIGNAL)}\ \frac{\begin{array}{c}\Gamma, ce \vdash e_{ck} : \{cr\} \mid \emptyset \qquad \Gamma, ce \vdash e_d : ct_2 \mid \emptyset \qquad \Gamma, ce \vdash e_g : ct_1 \xrightarrow{\emptyset+\star} ct_2 \xrightarrow{\emptyset+\star} ct_2 \mid \emptyset \\ \Gamma, ce \vdash e_{rck} : \{cr'\} \mid \emptyset \qquad \Gamma[x \mapsto gen_{cr}((ct_1, ct_2)\texttt{event}\{|cr|\}, e_{ck}, \Gamma)], ce \vdash e_1 : ct \mid cf_1\end{array}}{\Gamma, ce \vdash \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e_{ck}, \texttt{d} : e_d, \texttt{g} : e_g, \texttt{rck} : e_{rck})\ \texttt{in}\ e_1 : ct \mid cf_1 \cup \{cr\} \cup \{cr'\}}$$

$$\frac{\Gamma, ce \vdash e_s : (ct_1, ct_2)\texttt{event}\{|ce + \star|\} \mid \emptyset \qquad \Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_2 : ct \mid cf_2}{\Gamma, ce \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : ct \mid cf_1 \cup cf_2 \cup \{ce + \star\}}$$

$$\frac{\Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_s : (ct_1, ct_2)\texttt{event}\{|cr|\} \mid \emptyset \qquad \Gamma[x \mapsto ct_2], ce \vdash e_2 : ct \mid cf_2}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2 : ct \mid cf_1 \cup cf_2 \cup \{cr\}}$$

$$\frac{\Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_s : (ct_1, ct_2)\texttt{event}\{|ce + \star|\} \mid \emptyset}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{when}\ e_s : ct \mid cf_1 \cup \{ce + \star\}}$$

$$\text{(DOMAIN)}\ \frac{\Gamma[x \mapsto \forall \rho.\ \{\gamma + \rho\}], \gamma \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_b : \texttt{int} \mid \emptyset \qquad \gamma \notin ftv(\Gamma, ct)}{\Gamma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ e_b\ \texttt{do}\ e_1 : ct \mid cf_1 \setminus \{\gamma\}}$$

FIGURE 8.3 – Calcul d'horloges avec rangées

une rangée d'effets ouverte contenant un effet correspondant à la rangée d'horloges ρ de l'horloge prise en argument. Cela signifie que cette opération a un effet au moins sur l'horloge prise en argument.

La figure 8.3 montre les règles du système de types avec les rangées. Les règles manquantes sont les mêmes que dans le calcul d'horloges du chapitre 6 (figure 6.2). On peut faire les remarques suivantes sur les règles modifiées :

- Dans le cas de l'abstraction, on calcule d'abord le type ct_1 et l'effet cf_1 du corps. L'effet de la fonction est alors une rangée $cf_1 + fr$ contenant l'effet du corps.
- L'effet d'une application est la rangée d'effet fr stockée dans le type de la fonction e_1.
- Comme pour l'abstraction, l'effet d'une définition de processus est une rangée d'effets contenant l'effet cf_1 du corps. On utilise l'horloge d'activation ce' du processus pour typer son corps, comme dans le calcul d'horloges du chapitre 6. L'effet du lancement d'un processus est la rangée d'effets fr stockée dans le type du processus.
- Le type de l'expression `local_ck` est un type singleton associé à une rangée ouverte contenant l'horloge locale ce, ce qui permet donc de l'unifier avec d'autres horloges. On peut donc utiliser une horloge comme n'importe quelle valeur.
- Dans le cas de la déclaration de signal, l'horloge du signal est toujours donnée par le type de l'expression e_{ck}, mais c'est maintenant une rangée d'horloge cr. Pour forcer la fonction

de combinaison e_g du signal à ne pas avoir d'effet, sa rangée d'effets doit être fermée et ne contenir que l'effet vide. Les rangées fermées permettent donc d'imposer une borne maximale sur l'effet, alors que les rangées ouvertes expriment plutôt une borne inférieure.
L'originalité de cette règle est que l'on généralise uniquement les variables de rangées de l'horloge du signal dans le type de x. En particulier, si l'on omet l'horloge du signal, on a alors e_{ck} = `local_ck` et $s : \forall\rho.\ (ct_1, ct_2)\texttt{event}\{|ce + \rho|\}$. La variable de rangée ρ de l'horloge du signal est ainsi quantifiée universellement. Cela permet d'éviter les problèmes d'échappement de portée si jamais on unifie le type de s avec celui d'un signal sur une horloge plus rapide. Supposons par exemple que l'on unifie le type de s avec $(ct_1, ct_2)\texttt{event}\{|\gamma + \rho'|\}$ où γ est une horloge plus rapide. On devrait alors choisir $\rho \leftarrow \gamma + \rho_2$. L'horloge γ n'apparait pas dans le type de s car on a instancié le schéma de type de s par une variable de rangée ρ fraiche. Si la variable de rangée de s n'était pas généralisée, alors γ apparaitrait dans le type de s et échapperait donc de son domaine.

– Dans le cas de `present`, on doit forcer l'horloge du signal à être égale à l'horloge locale ce. Puisqu'un signal est maintenant paramétré par une rangée d'horloge, il faut s'assurer que cette rangée est fermée et réduite à une seule horloge égale à l'horloge locale. Elle doit donc être de la forme $ce + \star$.
– Dans le cas du `do`/`until`, la rangée d'horloges cr du signal e_s n'est plus contrainte. Il suffit de l'ajouter dans l'effet. La règle DOMAIN assure ensuite qu'il s'agit bien d'une rangée ne contenant que des horloges accessibles.
– Dans le cas d'un domaine réactif, la variable x correspondant à l'horloge locale a un type singleton dans lequel la variable de rangée ρ est quantifiée universellement. Les différentes occurrences de la variable x peuvent donc avoir des types différents.

Remarque 11. Le gain d'expressivité du calcul d'horloges avec rangées n'est pas directement visible dans les règles. Il vient des possibilités accrues d'unification des horloges et effets. Par exemple, l'unification des types de deux signaux peut échouer dans le calcul d'horloges normal s'ils ont des horloges différentes. Grâce à l'introduction des rangées d'horloges, on pourra leur donner la même rangée d'horloges et le même type.

Remarque 12. On peut noter que l'on obtient du sous-typage en combinant les variables de rangées avec le polymorphisme. Si jamais la variable de rangée d'une variable n'est pas universellement quantifiée, alors toutes les utilisations de la variable ont exactement le même type. Cela diminue donc la précision du système de types (mais pas sa correction), puisque l'on calcule une rangée d'effets ou d'horloges trop grande. C'est particulièrement un problème dans notre cas puisque l'on empêche l'échappement de portée. On peut donc se retrouver à détecter un échappement de portée simplement parce que notre version du sous-typage n'est pas assez précise. C'est pourquoi il faut que les variables de rangées des horloges et des signaux soient quantifiées universellement.

Syntaxe concrète

La forme générale d'un type devient maintenant :

`(ct1, .., ctn)` *ident*`{ c1, .., cm | cr1, .., crp | fr1, .., frq }`

Il est donc paramétré par une liste de types `ct`, une liste d'horloges `c`, une liste de rangées d'horloges `cr` et une liste de rangées d'effets `fr`. Le type d'un signal est ainsi `(ct1, ct2) event{|cr|}`, où `ct1` est le type des valeurs émises, `ct2` le type de la valeur reçue et `cr` est la rangée de l'horloge du signal. Celui d'un processus est `t process{c||fr}`, où `c` est son horloge d'activation et `fr` sa rangée d'effets. On note toujours `'c` les variables d'horloge et `?ck` la variable fraiche associée à l'horloge `ck`. On note + l'union des effets et <cr> l'effet associé à la rangée d'horloges `cr`. `[cr]` est le type singleton associé à la rangée d'horloges `cr`. On note également `'fr` (resp. `'cr`) les variables de rangées d'effets (resp. de rangées d'horloges) et `;` l'opérateur de concaténation de rangées. On note `..` les variables de rangées qui n'apparaissent qu'une seule fois dans un type. Enfin, la rangée fermée contenant uniquement l'effet `eff` est notée simplement `eff`.

Comme dans le calcul d'horloges, on note `ct1 =>{fr} ct2` le type des fonctions, qui contient maintenant une rangée d'effets `fr`. La notation `ct1 -> ct2` est maintenant un raccourci pour le

type `ct1 =>{0} ct2`, c'est-à dire pour une fonction paramétrée par une rangée d'effets fermée contenant uniquement l'effet vide, que l'on note $ct_1 \xrightarrow{\emptyset+\star} ct_2$ en syntaxe abstraite.

Exemples

Le calcul d'horloges avec rangées permet d'accepter des programmes dans lesquels on manipule des fonctions avec des effets différents. On peut par exemple revenir à notre mini-bibliothèque d'agents (figure 2.3 du chapitre 2), qui contient la fonction suivante :

```
let process proxy react msg = match msg with
| Mmsg msg -> run react msg (fun _ -> ())
| Mreq (msg, s) -> run react msg (fun v -> emit s v)
```

Cette fonction est rejetée par le calcul d'horloges sans rangées, car on appelle la fonction `react` avec la fonction `(fun _ -> ())` qui n'a pas d'effet et la fonction `(fun v -> emit s v)` qui a un effet sur l'horloge `'_c0` de `s`. Le calcul d'horloges requiert que les effets de ces deux fonctions soient égaux, ce qui n'est pas le cas :

```
This expression has clock '_a =>{'_c0} unit, but is used with clock '_a =>{0} unit
```

Elle est acceptée dans l'extension du calcul d'horloges que l'on vient de présenter :

```
val proxy ::
  ('a =>{'fr0} ('b =>{0;<'cr0>;..} unit) =>{'fr1} 'c process {''c0||'fr2}) =>{0;..}
  ('b , 'a) msg{|'cr0|} =>{0;..}
  'c process {''c0||{'fr3} + {'fr0} + {'fr1} + {'fr0} + {'fr3} + {'fr1};..}
```

Cette possibilité de stocker ensemble des fonctions avec des effets différents permet d'avoir un style de programmation plus naturel, sans avoir à se soucier des effets des fonctions. On peut par exemple simplifier l'écriture de la fonction `compute` (page 47) tirée de l'exemple des n-corps avec choix de la méthode d'intégration :

```
let process compute env x_t v_t w =
  let compute =
    match last current_imethod with
    | Pause -> compute_pause
    | Euler -> compute_euler
    | EulerSemi -> compute_euler_semi_implicit
    | Heun -> compute_heun
    | RK4 -> compute_rk4
  in
  run compute env x_t v_t w
```

Ce processus aurait été rejeté sans rangées puisque le processus `compute_pause` n'a pas d'effet, alors que les autres processus ont un effet sur l'horloge locale et l'horloge globale.

Notre extension du calcul d'horloges permet également de stocker ensemble des signaux d'horloges différentes. Un exemple d'utilisation est fourni par le processus `server` (page 19) du chapitre 2, qui reçoit sur un signal `add` un processus `p` et un signal `ack` sur lequel renvoyer la réponse :

```
let rec process server add =
  await add(p, ack) in
  run server add
  ||
  let v = run p in emit ack v
```

On peut désormais utiliser ce processus avec des processus et des signaux sur des rythmes différents :

exemples/ch8/server.rml

```
let process server_main =
  signal ack_slow in
  domain ck do
    signal add default (process (), ack_slow) gather (fun x _ -> x) in
    run server add
    ||
    emit add (process (pause global_ck; do_stuff ()), ack_slow);
    await ack_slow(_) in print_endline "Slow"
    ||
    pause global_ck;
    signal ack_fast in
    emit add (process (pause ck; do_stuff ()), ack_fast);
    await ack_fast(_) in print_endline "Fast"
  done
```

On émet d'abord sur `add` le signal lent `ack_slow` et un processus avec un effet sur l'horloge globale. On émet ensuite un signal rapide et un processus avec un effet sur l'horloge locale `ck`. Il est capital ici que la variable de rangée du signal `ack_slow` soit généralisée, comme on l'a vu dans la règle SIGNAL. Sinon, `ack_slow` aurait le même type que `ack_fast`, et donc l'horloge locale `ck` échapperait de son domaine.

Enfin, les rangées permettent maintenant d'utiliser vraiment les horloges comme des valeurs de première classe. On peut par exemple écrire un combinateur `run_one_step` qui lance son argument `p` pour un instant de l'horloge `ck` donnée en argument. Cette horloge peut être égale à `None`, auquel cas on interrompt le processus à la fin de l'instant de l'horloge globale :

```
let process run_one_step p ck =
  let ck = match ck with
    | None -> global_ck
    | Some ck -> ck
  in
  signal s clock ck in
  do run p until s done
  ||
  emit s
val run_one_step : unit process {''c0||'er0} =>{0;..}
  [ ?topck;.. ] option =>{0;..} unit process {''c0||<?topck;..> + {'er0};..}
```

On voit que le type de `ck` est un type singleton dont la rangée d'horloges est ouverte et contient au moins l'horloge globale `?topck`.

Implémentation

Il faut appeler le compilateur avec l'option `-row-clocking` pour utiliser le calcul d'horloges avec rangées. Son implémentation suit le principe de l'analyse de réactivité avec rangées que l'on a décrite dans la partie 8.2. La nouveauté est la présence de rangées ouvertes et fermées qu'il faut prendre en compte dans l'algorithme d'unification. Il faut remarquer que les rangées fermées que l'on manipule ne contiennent qu'un seul élément. Toutes les rangées d'effets fermées qui apparaissent au cours du typage ne contiennent que l'effet vide, alors que les rangées d'horloges fermées ne contiennent que l'horloge locale dans le cas des dépendances instantanées. L'unification se base donc sur l'invariant que chaque rangée fermée ne contient qu'un seul élément ou alors plusieurs copies du même élément. L'unification des rangées reste donc simple et il n'est pas nécessaire de comparer les effets ou horloges. Nous allons présenter ici uniquement l'unification de rangées d'effets, puisque l'on peut suivre le même raisonnement pour les rangées d'horloges. L'algorithme d'inférence suit le même principe que celui de la figure 8.1.

$$
\begin{aligned}
\mathcal{U}_{fr}(fr, fr) &= id \\
\mathcal{U}_{fr}(\psi, fr) = \mathcal{U}_{fr}(fr, \psi) &= \begin{cases} [\psi \leftarrow \mu\psi.\, fr] & \text{si } \texttt{occur\_check}(\psi, fr) \\ [\psi \leftarrow fr] \end{cases} \\
\mathcal{U}_{fr}(fr_1 + \psi_1, fr_2 + \psi_2) &= [\psi_1 \leftarrow fr_2 + \psi; \psi_1 \leftarrow fr_1 + \psi] \qquad \psi \text{ frais} \\
\mathcal{U}_{fr}(\mu\psi_1'.\, fr_1 + \psi_1, fr_2) = \mathcal{U}_{fr}(fr_2, \mu\psi_1'.\, fr_1 + \psi_1) &= \texttt{let}\ F_1 = \mu\psi_1'.\, fr_1 + \psi_1 \\
&\quad \texttt{in}\ \mathcal{U}_{fr}(fr_1[\psi_1' \leftarrow F_1] + \psi_1, fr_2) \\
\mathcal{U}_{fr}(fr_1 + \star, fr_2 + \star) &= \mathcal{U}_{cf}(\texttt{one}(fr_1), \texttt{one}(fr_2)) \\
\mathcal{U}_{fr}(fr_1 + \psi, fr_2 + \star) = \mathcal{U}_{fr}(fr_2 + \star, fr_1 + \psi) &= \texttt{let}\ \theta_1 = \mathcal{U}_{cf/fr}(\texttt{one}(fr_2), fr) \\
&\quad \texttt{let}\ \theta_2 = \mathcal{U}_{fr}(\psi, \star) \\
&\quad \texttt{in}\ \theta_2\theta_1
\end{aligned}
$$

(a) Unification des rangées d'effets

$$
\begin{aligned}
\mathcal{U}_{cf/fr}(cf, \star) &= id \\
\mathcal{U}_{cf/fr}(cf, cf') &= \mathcal{U}_{cf}(cf, cf') \\
\mathcal{U}_{cf/fr}(cf, fr_1 + fr_2) &= \texttt{let}\ \theta_1 = \mathcal{U}_{cf/fr}(cf, fr_1) \\
&\quad \texttt{let}\ \theta_2 = \mathcal{U}_{cf/fr}(\theta_1 cf, \theta_1 fr_2) \\
&\quad \texttt{in}\ \theta_2\theta_1
\end{aligned}
$$

(b) Unification d'un effet avec une rangée d'effets

FIGURE 8.4 – Unification des rangées d'effets

Pour pouvoir définir l'algorithme d'unification, on définit une opération qui prend une rangée d'effets et renvoie un des effets qu'elle contient, que l'on note $\texttt{one}(fr)$:

$$\texttt{one}(cf + fr) \triangleq cf$$

Cette opération ne sera utilisée que dans le cas des rangées fermées, où l'on sait que tous les éléments de la rangée sont identiques. La fonction est donc bien définie.

L'unification de rangées d'effets est définie sur la figure 8.4. Les quatre premiers cas sont les mêmes que pour l'unification de rangées de comportements, sur la figure 8.2b. Les deux nouveaux cas concernent les rangées fermées :

- Si on unifie deux rangées fermées, il suffit d'unifier un effet de la première avec un effet de la deuxième, puisqu'on sait que chaque rangée contient plusieurs copies du même effet (qui est l'effet vide en l'occurrence).
- Pour unifier une rangée ouverte $fr_1 + \psi$ avec une rangée fermée $fr_2 + \star$, on substitue la variable de rangée ψ par la rangée vide $\star$, puis on unifie chacun des éléments de la rangée fr_1 avec l'unique effet de la rangée fermée fr_2, que l'on obtient avec la fonction $\texttt{one}$ définie précédemment.

On utilise pour cela une fonction auxiliaire notée $\mathcal{U}_{cf/fr}$ qui prend en entrée un effet cf et une rangée d'effets fr et renvoie la substitution θ qui permet de rendre tous les effets contenus dans fr égaux à cf :

$$\mathcal{U}_{cf/fr}(cf, fr) = \theta$$

Cet algorithme est défini sur la figure 8.4b :

- Si la rangée est vide, alors on renvoie la substitution vide id.

– S'il s'agit d'un effet cf', alors on l'unifie avec l'effet cf en utilisant l'algorithme $\mathcal{U}_{cf}$ d'unification d'effets.
– Pour une rangée $fr_1 + fr_2$, on applique la fonction récursivement sur les deux composantes de la rangée.

Limites

Le calcul d'horloges avec rangées remplit bien son rôle puisqu'il permet d'accepter beaucoup plus de programmes. Il devient très rapidement nécessaire de l'utiliser à la place du calcul d'horloges sans rangées dès que l'on utilise l'ordre supérieur ou les signaux comme objets de première classe. La contrepartie de cette expressivité accrue est la complexité du système de types et des signatures de type. Le système de types comporte cinq sortes de types et quatre sortes de variables de type. Les types sont donc paramétrés par quatre sortes de variables, que l'utilisateur doit différencier au moment de la définition du type. Il n'est pas aisé de comprendre pourquoi le système de types contient à la fois des variables d'horloges, qui doivent être utilisées pour les horloges d'activation des processus, et des variables de rangées d'horloges pour les horloges des signaux. Mais il arrive également que l'on utilise une variable d'horloge pour un signal si un processus dépend instantanément de ce signal. En résumé, le calcul d'horloges avec rangées permet d'accepter de nombreux programmes, mais sa complexité est un obstacle à son utilisation par des utilisateurs non experts. La recherche d'un système de types plus simple mais permettant d'arriver au même résultat est donc un axe de recherche à poursuivre.

L'autre limite du système de types est liée à l'interaction avec le polymorphisme de rang supérieur, qui requiert l'écriture d'annotations de typage pour les arguments de fonctions polymorphes :

```
let process run_domain (q : 'ck. 'a process{'ck||'ck}) =
  domain ck do run q done
```

Comme dans le cas du calcul d'horloges normal, on ne peut décrire qu'un processus ayant un effet égal à l'horloge d'activation du processus. On pourrait penser que la notion de rangées permettrait d'exprimer le fait qu'une fonction a *au moins* un effet sur une certaine horloge, en écrivant :

```
let process run_domain (q : 'ck. 'a process{'ck||<'ck>; ..}) = ...
```

ce que l'on écrit $(q : \forall\gamma.\ \alpha\ \texttt{process}\{\gamma|\{\gamma + \star\} + \psi\})$ dans la syntaxe abstraite. Supposons qu'on ait alors un processus $p : \forall\gamma'.\ \alpha\ \texttt{process}\{\gamma'|(\{\gamma' + \star\} \cup \{\top_{ck} + \star\}) + \psi'\}$ correspondant au profil recherché, c'est-à-dire avec un effet sur son horloge d'activation et sur l'horloge globale. Pour unifier ces deux schémas de type, on doit prendre $\psi \leftarrow (\{\gamma + \star\} \cup \{\top_{ck} + \star\}) + \psi''$ et $\psi' \leftarrow (\{\gamma' + \star\}) + \psi''$ avec ψ'' frais. L'unification échoue alors car la variable quantifiée universellement ne doit pas apparaître dans les types que l'on substitue aux variables libres des schémas de type lors de l'unification. Autrement dit, il faut qu'après l'unification la variable γ n'apparaisse qu'aux endroits où elle apparaissait dans le type de départ. On pourra voir [JVWS07] pour une description plus formelle de l'unification de schémas de type. Le calcul d'horloges avec rangées n'améliore donc pas la situation en ce qui concerne le polymorphisme de rang supérieur.

Quatrième partie

Implémentation et applications

CHAPITRE 9

Implémentation

Nous nous intéressons dans ce chapitre à l'implémentation de REACTIVEML. Nous commençons par rappeler les principes de l'implémentation séquentielle du langage, avant de présenter plusieurs évolutions. Nous montrerons comment étendre cette implémentation pour prendre en compte les domaines réactifs et les différentes nouveautés du langage. Ensuite, nous décrirons une implémentation parallèle en mémoire partagée de REACTIVEML basée sur le *vol de tâches* [HS08], puis nous l'étendrons au cas des domaines réactifs. Enfin, nous présenterons une implémentation parallèle et distribuée du langage avec domaines utilisant des processus lourds communiquant par envoi de messages. Dans les différents cas, nous présenterons des résultats expérimentaux permettant de juger de l'efficacité des approches choisies. Les différentes implémentations décrites dans ce chapitre sont disponibles dans le dossier `interpreter/` de l'archive fournie avec le manuscrit et disponible à l'adresse :

http://reactiveml.org/these_pasteur/rml-these.zip

9.1 Implémentation séquentielle de REACTIVEML

Nous commençons par rappeler les principes de l'implémentation séquentielle de REACTIVEML. On trouvera une description formelle et détaillée dans [Man06]. Nous montrerons ensuite plusieurs évolutions de cette implémentation.

Moteur d'exécution

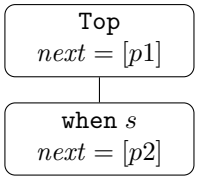
Le moteur d'exécution de REACTIVEML est un ordonnanceur de tâches légères. Il utilise un ordonnancement coopératif, où chaque processus doit volontairement rendre la main à l'ordonnanceur pour laisser les autres processus s'exécuter. On dispose d'une liste $\mathcal{C}$ de continuations à exécuter dans laquelle l'ordonnanceur vient piocher. Certains combinateurs, comme par exemple la composition parallèle, ajoutent des continuations dans cette liste. Il existe également une seconde liste appelée $next$ qui contient les processus à exécuter à l'instant suivant. On y ajoute typiquement la continuation de la construction `pause`. L'exécution d'un instant suit l'algorithme suivant :

1. On exécute un instant du programme. Pour cela, on exécute tous les processus dans la liste $\mathcal{C}$ jusqu'à ce qu'elle soit vide. Cela correspond à la réduction d'instant de la sémantique opérationnelle du chapitre 5.
2. On exécute ensuite la fin de l'instant. Comme pour la réduction de fin d'instant, on réveille les processus qui testent la présence d'un signal absent et ceux qui attendent la valeur d'un signal. On transfère ensuite les processus contenus dans la liste $next$, en attente de l'instant suivant, jusque dans la liste $\mathcal{C}$ des continuations à exécuter. On commence l'exécution de l'instant suivant en revenant à la première étape.

Le moteur d'exécution présente deux caractéristiques importantes qui viennent s'ajouter à cet algorithme de base :

- Pour obtenir un interprète efficace, il est capital de faire de *l'attente passive* des signaux. Cela signifie qu'un processus en attente d'un signal ne doit être activé que lorsque le signal est présent : il ne doit pas être réveillé pour vérifier la présence du signal plusieurs fois par instant, ni aux instants où le signal est absent. Pour éviter l'attente active, on associe à chaque signal une liste des continuations en attente de ce signal. On réveille ces continuations uniquement lorsque le signal est émis.
- L'autre composante du moteur d'exécution, qui est aussi la plus complexe, est la gestion des préemptions (`do`/`until`) et suspensions (`do`/`when`). C'est l'une des particularités qui distinguent REACTIVEML des autres bibliothèques de concurrence coopérative en OCAML comme LWT [Vou08] ou ASYNC [1]. L'utilisation d'une liste de continuations à exécuter fait que l'on perd complètement la structure du programme. Cela permet d'obtenir une exécution efficace, mais on a besoin d'une autre structure de données pour gérer l'activation des processus, puisque tous les processus ne sont pas forcément actifs à un instant donné. Cette structure s'appelle *arbre de contrôle*, puisqu'il s'agit d'un arbre n-aire traduisant l'imbrication des préemptions et suspensions dans le programme. Chaque nœud de cet arbre correspond à une construction **`do`**/**`until`** ou **`do`**/**`when`** dans le programme. Considérons par exemple le processus suivant :

```
let process control_tree s p1 p2 =
  emit s; pause; run p1
  ||
  do
    pause; run p2
  when s done
```



Le processus `control_tree` prend en entrée un signal `s` et deux processus `p1` et `p2`. Il lance `p1` et `p2` au second instant, mais `p2` n'est activé qu'aux instants où le signal `s` est présent. La figure sur la droite montre l'arbre de contrôle associé à l'exécution de ce programme à la fin du premier instant de l'exécution de `control_tree`. Sa racine notée `Top` correspond aux processus toujours actifs, alors que le nœud `when` s est associé à la suspension. Chaque nœud de l'arbre de contrôle contient une liste $next$ des continuations à exécuter au prochain instant où le corps de la structure de contrôle est actif, c'est-à-dire au prochain instant où le signal `s` est présent dans le cas de la suspension. La liste $next$ que l'on a évoquée précédemment est celle qui est associée au nœud `Top` et qui correspond aux processus activés à tous les instants. Ainsi, à la fin du premier instant, la liste $next$ de la racine de l'arbre contient le processus `p1`, alors que celle associée à la suspension contient `p2`. On transfère alors les processus depuis la liste $next$ de la racine dans la liste $\mathcal{C}$ des processus à exécuter, mais pas ceux du nœud `when` s. On ne le fera qu'à la prochaine émission du signal `s`, lorsqu'on saura que le corps de la suspension est activé.

Modularisation de l'interprète

Une des premières tâches que j'ai menée, du point de vue de l'implémentation, est la modularisation du moteur d'exécution. Ce travail a deux objectifs :

Partage de code Il existe plusieurs variantes du moteur d'exécution de REACTIVEML, qui diffèrent notamment par le choix de certaines structures de données. Elles ont cependant de très nombreux points communs. On souhaiterait donc partager le plus possible de code entre ces versions, mais aussi avec les versions parallèles du moteur d'exécution que nous présenterons dans la suite. Par exemple, le code du combinateur d'une boucle **`for`** parallèle, notée **`for`**/**`dopar`** en REACTIVEML, est le même quelle que soit la façon d'ordonnancer les continuations.

1. https://bitbucket.org/yminsky/ocaml-core/wiki/DummiesGuideToAsync

```
type 'a step = 'a -> unit

module type Runtime =
sig
  ...
  (* [on_current_instant f] execute f a l'instant courant *)
  val on_current_instant : unit step -> unit

  (* [on_next_instant ctrl f] execute f a l'instant suivant du noeud ctrl *)
  val on_next_instant : control_tree -> unit step -> unit

  (* [on_eoi f] execute f pendant la fin de l'instant *)
  val on_eoi : unit step -> unit

  (* [on_event ev ctrl f] execute f a l'emission de evt *)
  val on_event : ('a, 'b) event -> control_tree -> unit step ->

  (* [on_event_or_next ev f_ev ctrl f_n] execute f_ev si le signal est emis,
     sinon execute f_n a l'instant suivant *)
  val on_event_or_next : ('a, 'b) event -> unit step ->
     control_tree -> unit step -> unit
  ...
end
```

FIGURE 9.1 – Interface du module `Runtime` (@ `interpreter/runtime.mli`)

Séparation des problèmes (*separation of concerns*) Il s'agit d'un principe de base de génie logiciel. Le but est dans notre cas de distinguer le code des combinateurs, celui concernant l'ordonnancement des processus, ou encore les structures de données pour les files d'attentes de continuations. La séparation claire de ces composantes permet de faire des modifications beaucoup plus simplement.

Nous allons illustrer cette modularisation en nous intéressant plus particulièrement à l'ordonnancement des continuations. De nombreuses constructions du langage nécessitent d'attendre un certain événement, que ce soit la fin de l'instant, l'instant suivant ou encore la présence d'un signal. Nous allons voir qu'il suffit d'un petit nombre de primitives pour couvrir tous les cas de figures apparaissant dans le moteur d'exécution.

La figure 9.1 présente une partie de l'interface pour la gestion des continuations. On peut trouver la définition complète de ce module dans le fichier @ `interpreter/runtime.mli` de l'archive fournie avec la thèse. Celle-ci est un peu plus complexe puisqu'elle prend également en compte les domaines réactifs, mais sans que cela ne change le principe de l'interface. Le type d'une continuation qui attend une valeur de type `'a` est `'a step = 'a -> unit`. On définit ensuite plusieurs combinateurs correspondant aux différentes opérations dont on a besoin :

- `on_current_instant` exécute une continuation à l'instant courant. Cela revient à l'ajouter dans la liste $\mathcal{C}$.
- `on_next_instant` exécute une continuation à l'instant suivant où le nœud de contrôle est activé. Cela correspond typiquement à l'opérateur **`pause`** et revient à ajouter la continuation dans la liste $next$ du nœud de contrôle donné en argument.
- `on_eoi` exécute une continuation à la fin de l'instant. On l'utilise par exemple pour attendre la fin de l'instant pour connaître la valeur d'un signal.
- `on_event` exécute la continuation immédiatement après l'émission du signal, ce qui correspond à la construction **`await immediate`**.

– on_event_or_next correspond à la construction **present**. Il prend donc en entrée deux continuations. Il exécute la première immédiatement si le signal est émis et la seconde à la fin de l'instant dans le cas contraire.

A l'aide de ces opérations de base, on peut, par exemple, implémenter un combinateur appelé on_event_at_eoi qui exécute une continuation f à la fin de l'instant où le signal evt est émis. Cela correspond au comportement de la construction **await** (sans **immediate**) :

```
let on_event_at_eoi evt ctrl f =
  Runtime.on_event evt ctrl (fun () -> Runtime.on_eoi f)
```

Lorsque le signal evt est émis, on ajoute f dans la liste des continuations à exécuter à la fin de l'instant. On peut maintenant définir le combinateur rml_await_all', qui correspond à la construction **await** s(x) **in** e et qui permet de récupérer la valeur émise sur un signal :

```
let rml_await_all' evt p =
  fun f_k ctrl _ ->
    let await_eoi _ =
      let v = Runtime.Event.value evt in
      Runtime.on_next_instant ctrl (p v f_k ctrl)
    in
    on_event_at_eoi evt ctrl await_eoi
```

Ce combinateur prend en entrée un signal evt, un processus p à exécuter qui attend la valeur du signal, une continuation f_k, un nœud de l'arbre de contrôle ctrl et un dernier argument de type unit. On utilise le combinateur on_event_at_eoi, que l'on vient de définir, pour exécuter la fonction await_eoi à la fin de l'instant où le signal evt est émis. Cette fonction lit la valeur du signal en utilisant le module Runtime.Event qui regroupe les fonctions relatives aux signaux. Plus précisément, on appelle la fonction value qui renvoie la valeur du signal à l'instant courant. On demande enfin l'exécution du processus p à l'instant suivant où il est activé, en lui donnant comme argument la valeur v du signal, comme continuation f_k et comme nœud de l'arbre de contrôle ctrl. On peut trouver les définitions de ces deux combinateurs dans le fichier interpreter/lco_ctrl_tree_n.ml de l'archive. Elles diffèrent légèrement du fait de la gestion des domaines réactifs.

La définition des combinateurs implémentant les différentes expressions du langage (dans le fichier @ interpreter/lco_ctrl_tree_n.ml), qui est partagée entre toutes les implémentations, compte environ 800 lignes. La version séquentielle du moteur d'exécution (dans le fichier @ interpreter/seq_runtime.ml), c'est-à-dire le code qui n'est pas partagé, compte environ 750 lignes. Le code d'une version du moteur d'exécution dans l'ancienne version de REACTIVEML compte environ 1500 lignes. C'est donc la moitié du code que l'on peut partager. L'ajout des différents modules et interfaces implique une légère perte de performances, mais elle est négligeable par rapport au gain de clarté et aux possibilités d'évolution du moteur d'exécution. En plus de la modularisation, le nouveau moteur d'exécution utilise également une nouvelle méthode pour l'attente des signaux, que nous décrirons un peu plus loin, et prend en charge les domaines réactifs. Ces deux caractéristiques ont également un impact sur les performances. Il est donc compliqué d'isoler la perte de performances liée à la modularisation.

Attente passive avec préemption et suspension

Pour obtenir une implémentation efficace, on doit éviter l'attente active d'un signal. Lorsqu'un processus attend un signal, on le place donc dans la file d'attente attachée au signal et on le réveille uniquement lorsque le signal est émis. Mais si le processus s'exécute sous une suspension, il ne faut le lancer que si le signal de suspension est présent à cet instant. Dans le cas contraire, tout se passe comme si le processus n'avait pas vu l'émission et il doit rester en attente de la prochaine émission du signal. Considérons par exemple le processus suivant :

```
let process await_when =
  signal act, s in
  do
    await immediate s; print_endline "Recu!"
  when act done
  ||
  loop pause; emit act; pause end
  ||
  emit s; pause; emit s
```

On attend l'émission de `s` pour afficher immédiatement le message `"Recu"`, mais uniquement aux instants où le signal `act` est présent, c'est-à-dire aux instants pairs. La première émission de `s`, dans la troisième branche du parallèle, a lieu pendant le premier instant, au cours duquel le corps de la suspension n'est pas actif. On ne doit donc pas réveiller le processus en attente de `s`. On émet ensuite une seconde fois `act` au cours du deuxième instant. Comme le signal de suspension est cette fois présent, on affiche le message `"Recu"`.

Le choix fait dans l'implémentation actuelle de REACTIVEML est de ne pas gérer ce cas de figure. Elle utilise de l'attente active dans le cas où l'on attend un signal sous une suspension ou une préemption, c'est-à-dire qu'elle réveille le processus à chaque instant où son contexte est actif pour tester la présence du signal.

Nous avons choisi une autre approche permettant de gérer l'attente active dans tous les cas. On suit l'algorithme suivant pour lancer une fonction `f` lorsque le signal `evt` est émis et que le nœud de contrôle `ctrl` est actif (qui correspond à la fonction `on_event` de @ `interpreter/seq_runtime.ml`) :

1. Si le signal est présent à l'instant où on démarre l'attente, alors on peut lancer directement `f`. En effet, on sait que le nœud `ctrl` est actif puisque l'on est en train d'exécuter un processus dans ce contexte.

2. Sinon, on met une fonction dans la liste d'attente du signal. Celle-ci effectue les étapes suivantes à l'émission du signal :
   - Si le nœud de contrôle est actif, alors on lance `f`.
   - Sinon, on attend à la fois la fin de l'instant et l'activation du nœud de contrôle `ctrl` qui peut avoir lieu plus tard de l'instant. En effet, dans le cas d'une expression `do e when s done`, on active le nœud de contrôle correspondant à la suspension lorsque `s` est émis, ce qui peut se passer plus tard dans l'instant. Si le nœud `ctrl` est activé avant la fin de l'instant, alors on lance `f`. Sinon, on sait que le nœud n'est pas actif, ce qui signifie qu'il faut attendre la prochaine émission du signal `evt`. On recommence donc au début de l'étape 2.

L'inconvénient de cette méthode est qu'elle ajoute des traitements à chaque attente de signal. La version avec attente active est donc plus efficace si les processus sont très souvent actifs, alors que notre approche est plus avantageuse si les temps d'attente sont plus longs.

9.2 Implémentation séquentielle des domaines réactifs

Nous allons maintenant décrire l'implémentation séquentielle des domaines réactifs, disponible dans @ `interpreter/seq_runtime.ml`. Celle-ci ne pose pas de problème particulier et suit la sémantique opérationnelle du chapitre 5. C'est d'ailleurs un des avantages de cette nouvelle construction : puisqu'il s'agit d'une *réification* du moteur d'exécution de REACTIVEML, son implémentation requiert très peu de changements dans l'interprète. Cela s'explique par le fait que les fonctionnalités des domaines réactifs, comme l'attente automatique de l'horloge parente, sont fortement inspirées par les capacités du moteur d'exécution.

Principe

Domaines réactifs Un domaine réactif représente une instance du moteur d'exécution. Afin de pouvoir créer plusieurs domaines réactifs et les imbriquer, il faut regrouper toutes les variables globales du moteur dans une structure de données représentant l'état de l'interprète. Celle-ci contient la liste $\mathcal{C}$ des continuations à exécuter, la liste des fonctions en attente de la fin de l'instant et la racine de l'arbre de contrôle. On passe ensuite cette structure en argument des processus, tout comme on passe la pile courante dans la réduction d'instant. Toutes les fonctions du moteur, notamment les combinateurs de la figure 9.1, deviennent relatives au domaine réactif dans lequel s'exécute le processus.

Un domaine réactif est implémenté par un combinateur prenant en argument son corps, sous la forme d'une fonction attendant une horloge (voir la fonction `new_clock_domain` de `@ interpreter/seq_runtime.ml`). Cela permet d'expliquer pourquoi nous disons qu'un domaine réactif est une réification [FW84] du moteur d'exécution. En effet, il s'agit d'une construction du langage, représentée par une continuation dans l'implémentation, mais qui est elle-même un moteur d'exécution qui exécute les différentes phases que l'on a décrites précédemment.

L'exécution du domaine suit l'algorithme décrit dans la sémantique opérationnelle du chapitre 5 :

- On exécute un instant du domaine, ce qui correspond à la réduction d'instant (figure 5.2 page 77). Pour cela, on exécute toutes les continuations dans la liste $\mathcal{C}$ associée au domaine jusqu'à ce qu'elle soit vide.
- On effectue la fin de l'instant, qui correspond à la réduction de fin d'instant (figure 5.3 page 79). On réveille alors les processus qui attendent la fin de l'instant pour connaître la valeur d'un signal ou encore les tests de présence d'un signal absent.
- On décide ensuite s'il faut faire un autre instant local, ce qui est caractérisé dans la sémantique par le prédicat $\vdash_{\text{next}}$. Dans l'implémentation, il suffit de regarder s'il existe une continuation dans les listes $next$ des continuations à exécuter à l'instant suivant. On parcourt donc l'arbre de contrôle en partant du nœud associé au domaine. Si on trouve un processus dans la liste $next$ d'un nœud actif, alors on exécute un autre instant local, ce qui correspond à la règle LOCALEOI de la figure 5.2b. Sinon, ou bien si le compteur d'instants a atteint sa valeur maximale, il faut attendre la fin de l'instant de son horloge parente avant de passer à l'étape suivante (règle PARENTEOI).
- On effectue ensuite une nouvelle phase, que l'on appelle *passage à l'instant suivant*. Il s'agit de préparer l'exécution du prochain instant en transférant les processus depuis les listes $next$ actives dans la liste $\mathcal{C}$ associée au domaine réactif. Dans le cas des suspensions, on attend que le signal de suspension soit présent avant de faire ce transfert. Cela signifie que l'on transfère un processus dans la liste $\mathcal{C}$ lorsque l'on sait qu'il s'exécute dans un contexte d'évaluation Γ (défini page 55) ou que l'on peut appliquer la règle CONTEXTWHEN pour activer le corps d'une suspension (voir figure 5.2b page 77).

Dans le cas de REACTIVEML, la fin de l'instant et le passage à l'instant suivant sont confondus. En présence de domaines réactifs, lorsqu'un domaine est bloqué en attente de son domaine parent, il faut attendre la fin de l'instant de l'horloge parente pour effectuer le passage à l'instant suivant. En effet, un processus en attente du prochain instant du domaine parent ne sera réveillé qu'à la fin de l'instant du domaine parent, mais il ne faut pas oublier de l'ajouter à la liste $\mathcal{C}$ des processus à exécuter à l'instant suivant de l'horloge locale.

Signaux et attente passive On stocke maintenant dans chaque signal son horloge, qui est un lien vers la structure de données représentant l'état du domaine réactif correspondant. Il faut en effet attendre la fin de l'instant de l'horloge du signal pour connaître sa valeur pour l'instant courant. L'attente passive des signaux suit toujours le principe décrit dans la partie 9.1, mais il faut faire un peu plus attention. C'est en particulier le cas si on attend l'émission d'un signal lent, comme dans l'exemple suivant :

```
let process await_slow_when =
  signal s default 0 gather (+) in
  domain ck do
    signal act in
       loop emit act; pause ck; pause ck end
    || do await s(v) in print_int v when act done
    || pause ck; emit s 2
  done
```

La signal `act` est émis un instant sur deux, en commençant au premier instant. Lorsque le signal `s` est émis au cours du second instant de `ck`, le nœud de contrôle correspondant à la suspension n'est pas actif. Pourtant, le processus doit bien prendre en compte la présence du signal puisqu'il a été actif au cours du premier instant de `ck`, qui est inclus dans le même instant de l'horloge de `s`. On affiche donc bien un message à l'instant suivant de l'horloge de `s`.

Plus généralement, lorsqu'un signal est émis, il faut vérifier si le nœud de l'arbre de contrôle auquel est attaché le processus est actif ou a été actif au cours d'un des instants précédents inclus dans l'instant courant de l'horloge du signal (voir la fonction `has_been_active` du fichier @ `interpreter/seq_runtime.ml`). Pour cela, on stocke dans chaque nœud de l'arbre de contrôle les indices de toutes les horloges parentes au moment de sa dernière activation. Pour savoir si un processus peut voir l'émission du signal, il suffit alors de comparer l'indice courant de l'horloge du signal avec celui stocké dans le nœud correspondant à son contexte d'exécution.

Implémentation des autres constructions

Construction `qpause` La construction `qpause` se comporte de la même façon que la construction `pause`, mais elle doit être considérée comme en attente par le domaine réactif. Dans la sémantique opérationnelle, les deux constructions ne diffèrent que pour le prédicat $\vdash_{\text{next}}$ (voir figure 5.1 page 76). Dans l'implémentation, on va aussi les distinguer du point de vue de la décision de faire un autre instant local. On ajoute pour cela à chaque nœud de l'arbre de contrôle une liste $next\_control$. Les processus qui sont réellement en attente de l'instant suivant, comme `pause`, sont stockés dans $next$, alors que l'on stocke dans $next\_control$ ceux qui « ne comptent pas », comme `qpause`. On utilise aussi la liste $next\_control$ pour les fonctions auxiliaires comme celles que l'on utilise pour lancer l'activation du corps d'une suspension à l'émission du signal correspondant.

Après la fin de l'instant, on choisit de faire un autre instant local si la liste $next$ contient un processus, mais on ignore les processus de la liste $next\_control$. On transfère les continuations depuis les deux listes $next$ et $next\_control$ dans la liste $\mathcal{C}$ pour le passage à l'instant suivant.

Signal à mémoire On rappelle qu'un signal à mémoire, défini avec le mot-clé **`memory`** au lieu de **`signal`**, est un signal qui conserve sa valeur d'un instant au suivant (voir partie 2.3). Il suffit comme dans la sémantique de calculer la valeur du signal en combinant les valeurs émises en partant de la dernière valeur au lieu de la valeur par défaut (voir la définition de $S^v(n)$ page 58 et la fonction **`emit`** dans le fichier @ `interpreter/sig_env.ml`).

Réinitialisation des signaux Une autre nouveauté que l'on a introduite dans la partie 3.3 est la possibilité de réinitialiser la dernière valeur d'un signal à chaque instant d'une horloge `ck`, que l'on donne après le mot-clé **`reset`** à la définition du signal. Dans l'implémentation, on ajoute une fonction qui attend la fin de l'instant de `ck` pour réinitialiser le signal. Cette fonction utilise un pointeur faible (module `Weak` en OCaml) afin de ne pas empêcher le garbage collector de libérer le signal lorsque celui-ci n'est plus utilisé par le programme. On peut trouver le code correspondant dans la définition de la fonction `new_evt` du fichier @ `interpreter/seq_runtime.ml`.

9.3 Implémentation parallèle de REACTIVEML

Nous cherchons maintenant à répondre à la question de la parallélisation du moteur d'exécution de REACTIVEML. La version actuelle est une implémentation séquentielle de la concurrence avec un ordonnancement coopératif. On souhaite maintenant exécuter un programme REACTIVEML sur plusieurs cœurs pour gagner en efficacité, mais sans changer le programme et de façon transparente pour l'utilisateur.

Nous répondons à cette problématique en décrivant un moteur d'exécution parallèle en mémoire partagée de REACTIVEML. Il est implémenté dans le fichier `fsharp/rpml/rml_thread_runtime.fs` et compte environ 700 lignes de code. Nous commençons par un moteur d'exécution du langage sans domaines réactifs, puis nous verrons dans la partie suivante comment les prendre en compte. Nous utilisons des threads communiquant par mémoire partagée, ce qui permet d'effectuer des modifications minimales sur le moteur d'exécution.

Outils et langage

REACTIVEML est bâti sur un noyau d'OCAML (sans objets, variants polymorphes ni modules). Malheureusement, nous ne pouvons pas utiliser la version standard d'OCAML pour implémenter le moteur d'exécution parallèle. En effet, bien qu'il soit possible de créer des threads en OCAML (avec le module `Thread` de la librairie standard), seul un thread peut s'exécuter à la fois [2]. Cela s'explique par le fait que le moteur d'exécution d'OCAML, et en particulier le garbage collector, est séquentiel et ne peut pas être appelé depuis plusieurs threads en parallèle. Un verrou global empêche l'exécution de plusieurs threads OCAML en parallèle. Les travaux menés sur une extension parallèle du garbage collector d'OCAML [DL93] n'ont jamais été intégrés dans le langage, notamment car cela impacte les performances des programmes séquentiels [3]. Il existe plusieurs implémentations alternatives du langage sans cette limite, comme OCAML4MULTICORE [4] ou OCAML-JAVA [5], mais celles-ci sont très expérimentales.

Nous avons choisi d'utiliser le langage F# [6] pour mener à bien cette expérience. Il est en effet très proche d'OCAML et quasiment compatible au niveau du source avec le sous-ensemble d'OCAML que génère le compilateur REACTIVEML. Il existe également une bibliothèque de compatibilité avec OCAML [7] pour simplifier la transition. En ce qui concerne le sujet qui nous intéresse, le langage dispose d'un garbage collector parallèle, celui de la machine virtuelle .NET, ce qui permet de créer du code « vraiment » parallèle. L'intégration dans .NET permet d'accéder à une librairie standard riche, contenant tous les types de verrous usuels et des structures de données sans verrous (*lock-free*). Le langage est intégré à l'environnement de développement VISUAL STUDIO et dispose donc d'un *debugger* de qualité, compatible avec les programmes parallèles, et d'outils de *profiling* permettant notamment de visualiser l'exécution des threads (par exemple le temps passé à calculer ou à attendre un autre thread) et les conflits d'accès aux verrous. Ces caractéristiques ont été très utiles pour mener à bien ce projet.

Les principaux problèmes liés à l'utilisation de F# sont l'absence de foncteurs, que l'on doit remplacer par l'utilisation d'interfaces et de classes, mais surtout le fait que le langage ne puisse être utilisé que sous WINDOWS. Il existe une implémentation libre et multi-plateformes de .NET, appelée MONO [8], mais celle-ci présente des performances bien en deçà de celles de .NET et compte de nombreux bugs. Ainsi, la version parallèle que nous allons présenter ne fonctionne pas dans MONO. Le but de cette version parallèle est donc d'expérimenter la façon d'exécuter un programme REACTIVEML en parallèle et de voir les gains que cela pourrait apporter. Elle n'a pas vocation, en l'état, à devenir une implémentation de référence du langage.

2. Voir par exemple partie 19.10.2 *Parallel execution of long-running C code* de http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html
3. http://caml.inria.fr/pub/ml-archives/caml-list/2002/11/85c22e6a9db58c12c8a80d6d316c6622.en.html
4. http://www.algo-prog.info/ocmc/web/
5. http://ocamljava.x9c.fr/preview/
6. http://tinyurl.com/fsspec
7. Le module `FSharp.PowerPack.Compatibility` disponible à l'adresse http://fsharppowerpack.codeplex.com/
8. http://mono-project.com/Main_Page

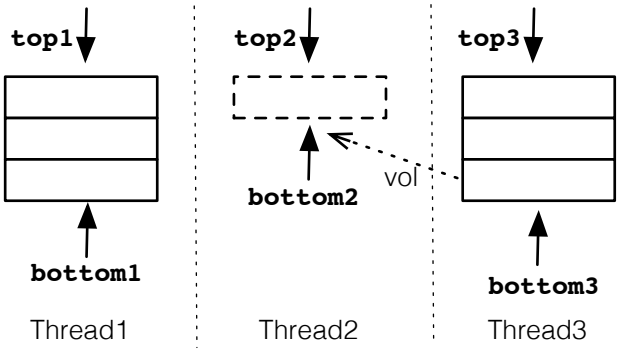


FIGURE 9.2 – Vol de tâches

Implémentation

On a vu que le moteur d'exécution de REACTIVEML est centré autour d'une liste $\mathcal{C}$ de continuations dans laquelle l'ordonnanceur vient piocher des tâches à exécuter. Le principe de la version parallèle en mémoire partagée est d'avoir plusieurs threads qui viennent piocher des continuations dans la liste $\mathcal{C}$ et les exécutent. On ne parallélise que l'exécution au cours d'un instant. En particulier, la fin de l'instant reste purement séquentielle et tous les threads se synchronisent à la fin de chaque instant.

Pour représenter la liste $\mathcal{C}$, on utilise une structure de données concurrente pour réaliser du *vol de tâches* ou *work stealing* [HS08]. La figure 9.2 représente le principe de l'algorithme. Chaque thread dispose de sa propre liste de continuations à exécuter, qui est une file à double entrée ou *deque*, symbolisée par un sommet noté `top` et un pointeur `bottom` indiquant le bas de la file. Lorsque la liste d'un thread est vide, comme dans le cas du second thread de la figure, il va « voler » des tâches à exécuter dans la file d'un autre thread, ici le troisième thread. L'utilisation d'une file à double entrée permet d'éviter dans la majorité des cas les conflits entre le voleur, qui accède au bas `bottom` de la file, et le propriétaire de la file qui accède au sommet `top` de la file. Cet algorithme peut s'implémenter de façon très efficace sans verrous avec des opérations atomiques comme le *compare-and-swap* [CL05]. Nous utilisons la classe `System.Collections.Concurrent.ConcurrentBag` de la librairie standard .NET qui implémente cette structure de données.

Puisque les threads communiquent par mémoire partagée, on peut réutiliser l'essentiel du code de la version séquentielle du moteur d'exécution. Il faut toutefois gérer les accès concurrents aux différentes structures de données du moteur d'exécution. Le principe est d'associer un verrou à chaque structure partagée. On peut cependant faire mieux :

- On associe à chaque composition parallèle synchrone un compteur du nombre de branches qui ont terminé leur exécution. Comme il s'agit d'entiers, on utilise des opérations atomiques pour décrémenter ces compteurs, de la classe `System.Threading.Interlocked`. Par exemple, la fonction `Interlocked.Decrement` décrémente une référence et renvoie sa nouvelle valeur de façon atomique. On peut donc se passer de verrou dans ce cas.
- On peut utiliser des structures de données sans verrous [HS08] pour les listes partagées. On utilise par exemple une file sans verrous de la classe `ConcurrentStack` pour les listes *next* des nœuds de l'arbre de contrôle.
- On peut se passer de verrous s'il n'y a pas de risques de course critique (*data race*). On parle de course critique si une lecture et une écriture ou deux écritures sur une même location mémoire ne sont pas ordonnées par des verrous. Dans le cas de la dernière valeur d'un signal, elle n'est modifiée que pendant la fin de l'instant, qui est séquentielle, et on ne fait que lire la valeur pendant l'instant. Il n'y a donc pas de risque de course critique. Il faut tout de même prendre garde que les valeurs écrites par un thread soient bien vues par tous les autres threads, mais cela ne pose pas de difficulté particulière dans le modèle mémoire de .NET [9].

9. http://msdn.microsoft.com/fr-fr/magazine/cc163715(en-us).aspx

On utilise des verrous pour les nœuds de l'arbre de contrôle, mais ceci ne pose pas de problème car les modifications de ces nœuds sont rares. L'autre cas où l'on utilise encore un verrou est pour l'implémentation des signaux. En effet, au moment de l'émission d'un signal, on doit, de façon atomique, changer le statut du signal et récupérer les processus en attente du signal pour les réveiller. Il faut utiliser un verrou pour garantir qu'aucun autre processus ne puisse voir le signal absent sans être dans la liste des processus en attente.

Résultats expérimentaux

La figure 9.3 montre les performances du moteur d'exécution parallèle écrit en F#. On mesure ici l'accélération par rapport à la version séquentielle F#, c'est-à-dire le rapport entre la durée du calcul pour la version séquentielle et la durée du calcul dans la version parallèle. Plus la barre est élevée, plus la version parallèle est rapide. Ainsi, une accélération de 1.02 signifie que la version parallèle est 2% plus rapide que la version séquentielle, alors qu'une accélération de 3.8 signifie que la version parallèle est 3,8 fois plus rapide. On mesure le temps d'exécution total de plusieurs instants de chaque programme.

La machine de test possède deux processeurs Xeon 5150 à 2,66 Ghz avec deux cœurs chacun. Elle tourne sous Windows 7 et on utilise Visual Studio 2012 Ultimate. Lorsque cela n'est pas précisé, on utilise autant de threads que de cœurs, c'est-à-dire 4. Les programmes sont compilés avec optimisation pour une architecture 64 bits. On teste les programmes en utilisant les deux garbage collector (GC) disponibles pour la machine virtuelle .NET, puisque l'on verra que cela a une grande influence sur les performances. Le GC *workstation* est celui qui est activé par défaut et il est conçu pour les applications interactives. Le GC *server* a moins de contraintes de latence, ce qui lui permet d'être plus efficace, en particulier dans un programme avec plusieurs threads. On peut voir une description plus détaillée des différences entre ces deux garbage collector dans la documentation de .NET [10].

Nous allons maintenant commenter chacun de ces graphes :

– Le premier graphe teste l'accélération dans le cas d'un programme simple :

```
let process test_compute  =
  for i=1 to 100 dopar
    loop do_stuff compute_intensity; pause end
  done
```

  La fonction `do_stuff` fait un calcul dont la durée est proportionnelle à son argument. Il s'agit d'une boucle **for** qui n'alloue aucune mémoire, car on cherche à mesurer l'efficacité de la méthode d'ordonnancement, sans influence du garbage collector. On affiche l'accélération suivant la valeur du paramètre `compute_intensity` qui est une entrée du programme que l'on représente en abscisses du graphe. Lorsqu'il est égal à zéro, on se retrouve dans le pire cas d'un programme où des processus ne font que se synchroniser sans jamais calculer. On peut remarquer que l'on obtient tout de même une exécution parallèle légèrement plus rapide, puisque l'on obtient une accélération de 1.02. Sans surprise, on remarque ensuite que l'accélération augmente avec la quantité de calcul. Ainsi, la version parallèle est 3,8 fois plus rapide lorsque les synchronisations sont rares.

– Le premier graphe montre que le vol de tâches est efficace, mais il ne dit rien sur l'efficacité du moteur d'exécution parallèle. On a donc testé plusieurs exemples typiques de REACTIVEML, tirés de la distribution du compilateur. On peut observer le résultat sur le second graphe. On fait varier le nombre de planètes dans le cas de la simulation des n-corps (figure 2.1 page 20) pour tester plusieurs rapports calcul/synchronisation. Les autres exemples sont des simulations d'automates cellulaires [Bou04b] (`exemples/cellular`) et de *boids* [Rey87] (`exemples/boids`), où l'on simule une nuée d'oiseaux en vol. On voit que l'on obtient de bons résultats avec une version parallèle deux fois plus rapides environ, sauf dans le cas de la simulation d'automates cellulaires. Cela peut s'expliquer par le fait que ce programme

10. http://msdn.microsoft.com/en-us/library/ee787088.aspx#workstation_and_server_garbage_collection

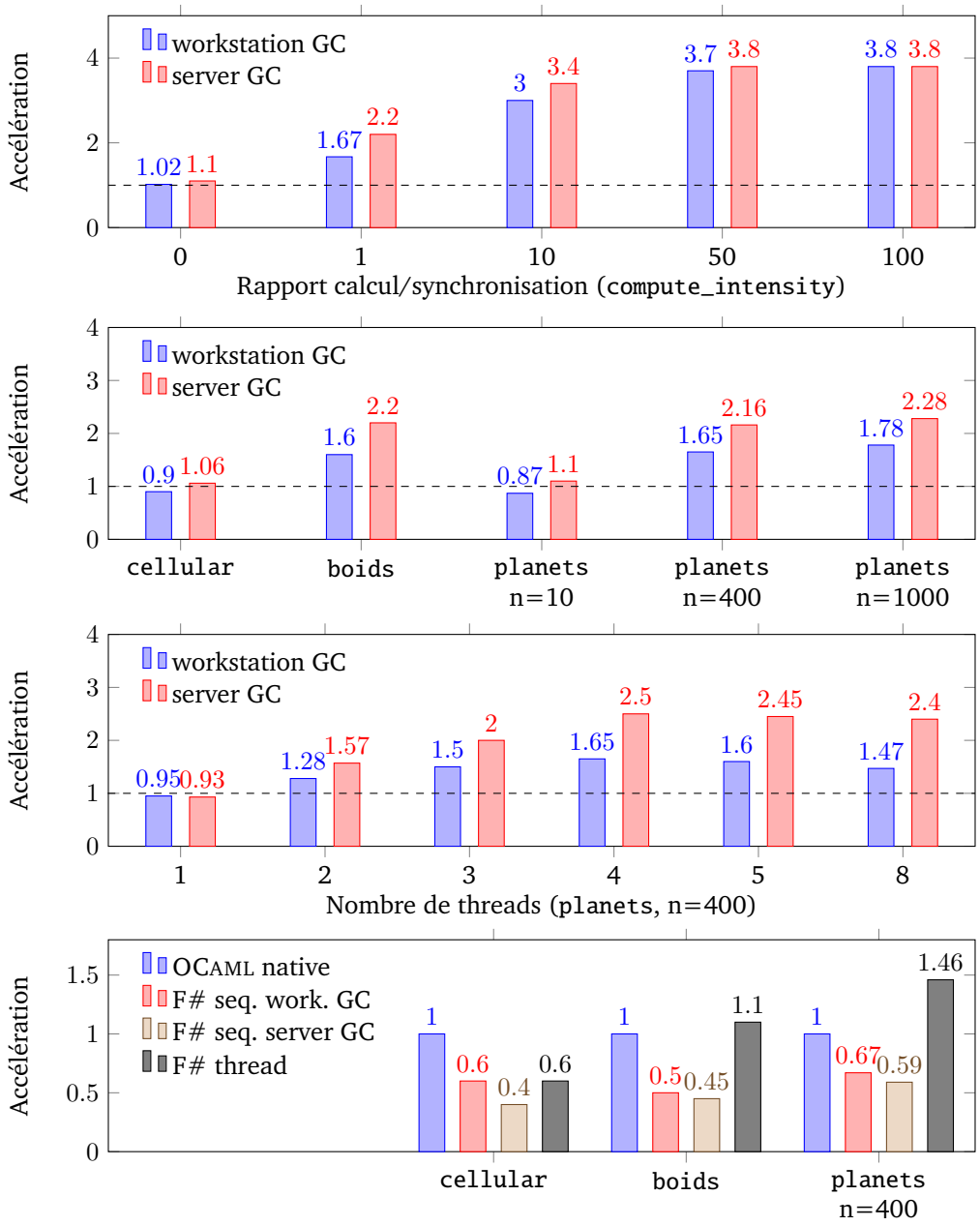


FIGURE 9.3 – Résultats expérimentaux du moteur d'exécution parallèle en F#

fait très peu de calculs pour chaque processus et aussi par le fait qu'il alloue énormément de mémoire (on alloue une continuation à chaque appel de **pause**), ce qui pose problème puisque le garbage collector de .NET bloque tous les threads.
- Le troisième graphe montre l'accélération en fonction du nombre de threads utilisés pour le calcul. Sans surprise, on obtient le meilleur résultat lorsque l'on a autant de threads que de cœurs sur la machine, c'est-à-dire 4.
- Enfin, le quatrième graphe mesure l'efficacité du moteur d'exécution par rapport à la version OCAML native, qui est cette fois la référence. On peut voir que la version F# séquentielle est toujours plus lente, souvent par un facteur assez grand. Par exemple, elle est deux fois plus lente dans le cas des automates cellulaires ou des boids. L'utilisation du parallélisme permet le plus souvent de combler ce retard, mais n'offre pas des gains importants par rapport à la version OCAML native de référence. Cela relativise donc l'utilité pratique de cette version du moteur d'exécution, en dehors des expérimentations sur la parallélisation du langage. L'autre remarque à faire sur ce graphe est que le GC *server* est notablement plus lent dans le cas séquentiel, alors qu'il devient bien plus rapide dans le cas parallèle. On voit donc que la question du garbage collector est centrale pour les performances d'une exécution parallèle d'un langage fonctionnel.

Remarque 13. Le moteur d'exécution parallèle que l'on vient de décrire présente une grande différence avec la version séquentielle si on utilise des références dans un programme REACTIVEML. En effet, la version séquentielle garantit l'atomicité des expressions ML pures. On peut donc accéder de façon concurrente à une référence comme dans cet exemple :

```
let process ref_concurrent =
  let r = ref 0 in
  (r := !r + 1 || r := !r - 1);
  print_int !r
```

L'ordre d'exécution des deux branches du parallèle est inconnu, mais on sait que les deux affectations à `r` sont faites de façon atomique, donc le programme affiche toujours 0. Cette propriété des références est utilisée par de nombreux programmes REACTIVEML pour la gestion d'un état partagé, comme on l'a déjà mentionné dans la partie 2.4. Elle n'est pourtant pas garantie dans la sémantique du langage étendu avec les références [Man06]. Elle devient même fausse dans le moteur d'exécution parallèle, à moins d'ajouter un verrou garantissant qu'une seule expression ML est évaluée à la fois. L'ajout de ce verrou global fait perdre tout l'intérêt de l'exécution parallèle et on obtient des performances proches de la version séquentielle. Nous n'avons donc pas conservé cette solution, ce qui implique que l'on ne peut pas tester les programmes REACTIVEML plus conséquents, comme les simulations de réseau ad-hoc [SMMM06], dans ce moteur. Il faudrait modifier ces exemples pour se passer de références. Nous ne l'avons pas fait par manque de temps.

9.4 Implémentation parallèle avec domaines réactifs

Nous allons maintenant étendre l'implémentation parallèle du moteur d'exécution pour prendre en compte les domaines réactifs. Cette nouvelle version du moteur est implémentée dans le fichier `fsharp/rpml/thread_runtime.fs` et compte environ 800 lignes. Nous avons vu dans la partie 3.1 que les domaines réactifs présentaient des avantages pour une exécution parallèle du code. En particulier, ils diminuent les synchronisations, puisqu'un domaine réactif peut exécuter plusieurs instants locaux indépendamment des autres domaines réactifs et processus. Cela permet donc d'augmenter la granularité du parallélisme, c'est-à-dire d'augmenter la part de calcul par rapport aux communications.

Principe

Le principe de cette version du moteur d'exécution est que chaque domaine réactif s'exécute dans un seul thread, avec un ordonnancement coopératif des processus comme dans la version séquentielle. Pour gérer l'équilibrage de charge entre les threads, on utilise encore le vol de tâches, mais cette fois les tâches ne correspondent plus à des processus mais à des domaines réactifs. Cela signifie que lorsque l'on veut exécuter un domaine réactif, on ajoute la tâche correspondante dans la liste globale, afin que le domaine puisse s'exécuter dans un autre thread en parallèle des autres processus et domaines réactifs.

L'avantage de cette approche est que les synchronisations sont plus rares, puisque l'on exécute tous les processus à l'intérieur d'un domaine réactif avant de se synchroniser. On peut en outre supposer que les accès concurrents sont plus rares, puisque les processus dans le même domaine réactif communiquent a priori plus souvent entre eux qu'avec le reste du programme. La contrepartie est que l'on obtient une exécution efficace seulement si le programme contient plusieurs domaines réactifs. C'est a priori le cas d'un programme REACTIVEML typique, où de nombreux agents, correspondant chacun à un domaine réactif, sont exécutés de façon concurrente. Si le programme REACTIVEML ne contient pas de domaines réactifs, les performances seront les mêmes que pour la version séquentielle.

Implémentation

L'implémentation parallèle des domaines réactifs réutilise, là encore, l'essentiel du code de la version séquentielle. Il faut toujours protéger les accès concurrents aux structures de données partagées, comme les signaux ou les nœuds de l'arbre de contrôle, par des verrous ou des opérations atomiques. Mais puisque tous les processus dans le même domaine réactif sont exécutés par le même thread, on n'a plus d'accès concurrents pour les compteurs de synchronisation de la composition parallèle synchrone. On peut donc utiliser un simple compteur sans opérations atomiques.

Pour gérer le fait que les domaines réactifs s'exécutent potentiellement dans un autre thread que leur domaine parent, il faut faire plusieurs modifications :

- On associe à chaque domaine un compteur du nombre de domaines enfants détachés en cours d'exécution.
- Si le domaine réactif a terminé d'exécuter tous les processus dans sa liste $\mathcal{C}$, il vérifie si ce compteur est à zéro. Si c'est le cas, alors tous les domaines réactifs enfants ont terminé leur exécution pour l'instant courant. Il peut donc exécuter la fin de l'instant local, comme dans la version séquentielle. Sinon, il doit attendre la fin de l'exécution d'un de ses domaines réactifs enfants. Il se met alors en sommeil.
- Lorsqu'un domaine réactif termine son exécution et attend le prochain instant de son domaine parent, il décrémente le compteur de son parent. Si celui-ci atteint zéro et que le domaine parent est en sommeil, alors on exécute la fin de l'instant du domaine parent dans le thread courant.

Il faut également prendre en compte le fait qu'un processus en attente d'un signal peut être réveillé suite à l'émission du signal dans un domaine réactif enfant, qui s'exécute donc dans un autre thread. On associe pour cela à chaque domaine réactif une seconde liste de continuations à exécuter, mais qui est cette fois une file sans verrous $\mathcal{F}$, alors que la liste $\mathcal{C}$ est une simple liste qui n'est jamais lue par plusieurs threads en même temps. On y ajoute les processus réveillés suite a l'émission d'un signal dans un autre thread. On y met également la continuation d'un domaine réactif enfant lorsqu'il termine, puisque celui-ci s'exécute dans un autre thread. Avant d'exécuter la fin de l'instant d'un domaine, on exécute tous les processus contenus dans la file $\mathcal{F}$. L'absence de dépendances instantanées sur un signal lent garantit que l'on peut faire cela une seule fois à la fin de l'instant, après que tous les domaines réactifs enfants aient terminé leur exécution.

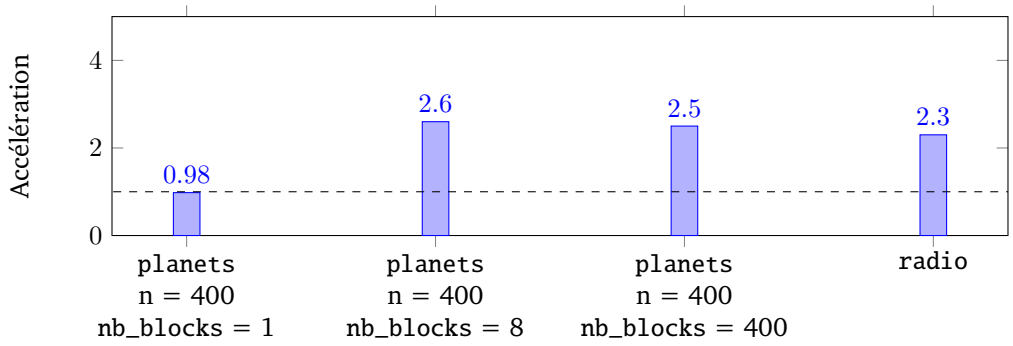


FIGURE 9.4 – Résultats expérimentaux du moteur d'exécution parallèle avec domaines réactifs

Résultats expérimentaux

Puisque ce second moteur d'exécution parallèle se base sur les mêmes principes que le premier, les résultats expérimentaux sont proches. On ne fait les tests ici qu'avec le GC server. On cherche à savoir si les exemples du chapitre 3 peuvent bénéficier de l'exécution parallèle.

La figure 9.4 montre l'accélération de la version parallèle pour plusieurs exemples. Le premier exemple reprend la simulation des n-corps de la figure 2.1 (page 20). On lance cette fois les processus correspondant aux planètes en les regroupant à l'intérieur de `nb_blocks` domaines réactifs :

```
let process main =
  ...
  ||
  for j=0 to nb_blocks-1 dopar
    domain ck by 1 do
      for i=1 to (nb_planets/nb_blocks) dopar
       run body (random_planet ())
     done
    done
  done
```

On utilise ici des domaines réactifs de période 1 pour répartir le code dans plusieurs threads. Cela ne change pas dans cet exemple la sémantique du programme et permet d'exécuter les domaines en parallèle. Dans le cas où `nb_blocks` = 1, on exécute un seul domaine réactif donc il n'y a aucun parallélisme pour ce moteur d'exécution. La version parallèle est légèrement plus lente à cause du surcout de la gestion des threads. Si `nb_blocks` = `nb_planets`, on a un domaine réactif par planète et on se retrouve donc dans une situation proche de celle du moteur d'exécution de la partie précédente, avec un code parallèle 2,5 fois plus rapide. Le point le plus important de ce graphe est le cas intermédiaire avec `nb_blocks` = 8. On obtient alors un programme plus rapide que dans le cas de la version parallèle sans domaines. En effet, on a regroupé plusieurs planètes dans un domaine réactif, ce qui augmente la quantité de calcul par tâche. On a ainsi augmenté le rapport calcul/synchronisation et donc les performances de l'exécution. Enfin, le dernier exemple est celui du capteur avec simulation de l'énergie de la figure 3.2 (page 43), dont les résultats sont donnés par la quatrième barre du graphe. On remarque que l'on obtient une très bonne accélération, puisque chaque agent simule mille instants locaux par pas de la simulation avant de se synchroniser.

9.5 Implémentation distribuée des domaines réactifs

Nous décrivons maintenant une implémentation parallèle et distribuée de REACTIVEML. Elle est implémentée dans le fichier @ `interpreter/distributed_runtime.ml` et utilise plusieurs modules auxiliaires définis dans le dossier `interpreter/mpi`, pour un total de 2500 lignes de code.

Plutôt que d'utiliser des threads communiquant par mémoire partagée, on utilise des processus lourds communiquant par envoi de messages, ce qui permet de traiter aussi bien le cas des processeurs multi-cœurs que l'exécution distribuée sur un réseau. C'est l'approche traditionnelle pour la programmation parallèle en OCaml, utilisée par exemple par les bibliothèques OCamlnet [11] et Functory [12].

Principe

Comme dans la partie précédente, on distribue le code au niveau des domaines réactifs. L'exécution au sein de chaque domaine est séquentielle avec ordonnancement coopératif. Chaque processus lourd exécute un ensemble de domaines réactifs et communique avec les autres processus par envoi de messages. Ce sont des valeurs OCaml sérialisées avec le module `Marshal` [13] de la bibliothèque standard.

La base du moteur d'exécution distribué est une boucle d'événements, qui appelle des *callbacks* à chaque réception d'un message d'un autre processus. Le processus principal lance l'exécution du domaine réactif global du programme, alors que les autres processus traitent les messages qu'ils reçoivent, puis se remettent en attente de nouveaux messages. Nous décrirons dans la partie suivante les différents types de messages envoyés par les processus et les traitements associés.

L'envoi de messages se fait en utilisant MPI [14] (*Message Passing Interface*). C'est un standard définissant un ensemble de primitives pour l'envoi de messages entre programmes, avec de nombreuses variantes selon que l'envoi et la réception sont bloquants ou non. Il existe de nombreuses implémentations de ce standard et nous utilisons MPICH [15]. Elle permet de communiquer par mémoire partagée sur une même machine ou par TCP sur un réseau, le tout de façon transparente pour le programmeur. L'utilisation de MPI simplifie aussi le déploiement des programmes, puisque la bibliothèque se charge de lancer les différents processus localement ou sur une machine distante.

Nous avons écrit une petite bibliothèque pour appeler les fonctions MPI depuis OCaml. Elle se base sur la bibliothèque OCaml/MPI [16] que l'on a adaptée à nos besoins. Le code de cette bibliothèque est disponible dans le dossier `mpi/` de l'archive distribuée avec le manuscrit. Il compte environ 600 lignes de C et 200 lignes d'OCaml.

Protocole

Gestion des domaines réactifs Le tableau suivant présente les différents messages associés à la gestion des domaines réactifs, le type du contenu du message et une courte description de son utilisation.

Nom	Contenu	Description
`Mnew_cd`	`t`	Création d'un domaine
`Mstep of gid`	`unit`	Exécution d'un instant du domaine
`Mstep_done of gid`	`unit`	Instant terminé
`Mdone of gid`	`'a`	Domaine réactif terminé
`Meoi`	`gid`	Fin de l'instant du domaine d'horloge
`Mhas_next of gid`	`bool`	Réponse pour le calcul de la fin du macro instant
`Mnew_remote of gid`	`site`	Gestion des sites distants

où `t = gid * clock * (clock_domain -> control_tree -> unit step)`. Un message paramétré par le type `gid` des identifiants de domaines réactifs est destiné à un domaine particulier, alors que les autres messages sont destinés au processus.

11. http://projects.camlcity.org/projects/ocamlnet.html
12. http://functory.lri.fr/About.html
13. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html
14. http://www.mcs.anl.gov/research/projects/mpi/
15. http://www.mpich.org/
16. http://forge.ocamlcore.org/projects/ocamlmpi/

L'exécution des domaines réactifs suit le même algorithme que dans la version séquentielle que l'on a décrite dans la partie 9.2. Le principe de l'implémentation distribuée est de remplacer chaque phase de l'exécution par un envoi de messages lorsque le domaine réactif s'exécute dans un autre processus que son domaine parent.

Le message `Mnew_cd` demande la création d'un nouveau domaine réactif sur un processus distant. Le message contient un identifiant temporaire pour le domaine, l'horloge du domaine parent et la fonction correspondant au corps du domaine, paramétrée par le domaine et le nœud de l'arbre de contrôle correspondant. A la réception de ce message, le processus crée un nouveau domaine réactif puis l'exécute. Si le domaine réactif termine, il envoie le message `Mdone` contenant sa valeur de retour à son domaine parent, qui va alors lancer sa continuation. Sinon, le domaine envoie un message `Mstep_done` pour signaler qu'il a terminé son exécution pour l'instant courant. Il attend la réception de `Mstep` pour exécuter les prochains instants locaux du domaine.

Les autres messages concernent la gestion de la fin de l'instant. Le message `Meoi` indique qu'il faut exécuter la fin de l'instant de l'horloge contenue dans le message. Ce message est envoyé lorsque le domaine a terminé l'exécution des processus locaux et que tous ses domaines enfants ont envoyé un message `Mstep_done` ou `Mdone`. Chaque domaine réactif enfant va en réponse à `Meoi` envoyer un message `Mhas_next` qui indique si ce domaine doit exécuter un autre instant de l'horloge dont c'est la fin de l'instant. Cette étape remplace le parcours de l'arbre de contrôle dans la version séquentielle. Le passage à l'instant suivant se fait au moment de la réception du message `Mstep` à l'instant suivant.

Enfin, le message `Mnew_remote` permet à chaque domaine réactif de maintenir à jour une liste contenant l'ensemble des processus dans lesquels s'exécute un domaine réactif enfant. C'est à ces sites qu'il faut envoyer les messages `Meoi` signalant la fin de l'instant de l'horloge du domaine.

Gestion des signaux

Nom	Contenu	Description
`Mcreate_signal`	`gid * (unit -> unit)`	Création d'un signal distant
`Msignal_created of gid`	`('a, 'b) event`	Signal distant créé
`Mreq_signal of gid`	`unit`	Demande de la valeur du signal
`Msignal of gid`	`('a, 'b) event`	Valeur actuelle du signal
`Memit of gid`	`'a`	Émission d'une valeur
`Mvalue of gid`	`'b`	Nouvelle valeur du signal

Chaque signal est géré par le processus où s'exécute le domaine réactif associé à son horloge. Si on veut créer un signal associé à un domaine exécuté sur un autre processus, il faut envoyer un message `Mcreate_signal` pour demander la création du signal. On envoie avec ce message la fonction qui va créer le signal, car le processus distant ne connait pas le type du signal à créer. En réponse, on reçoit le signal avec un message `Msignal_created`.

Chaque processus conserve une copie locale des signaux auxquels il accède. Il faut envoyer un message `Mreq_signal` au domaine qui gère le signal pour obtenir la valeur actuelle du signal. On reçoit alors un message `Msignal` contenant la valeur correcte du signal que l'on peut stocker dans un cache local. Lors des accès ultérieurs, on utilise directement la copie locale du signal.

Enfin, l'émission sur un signal géré par un domaine réactif distant se fait par l'envoi d'un message `Memit`. A la réception de ce message, le propriétaire du signal combine la valeur reçue avec la valeur courante du signal. Pour maintenir à jour la valeur des copies locales des signaux, le propriétaire d'un signal envoie à la fin de chaque instant où le signal est émis un message `Mvalue` contenant la valeur du signal à cet instant. Chaque processus met à jour sa copie locale en utilisant cette valeur. L'impossibilité de dépendre instantanément d'un signal lent permet de ne communiquer la valeur d'un signal qu'une seule fois par instant et de sinon accéder à la copie locale.

Remarque 14. Après l'envoi d'un message `Mreq_signal` pour obtenir la valeur d'un signal, le processus bloque jusqu'à la réception du message `Msignal`. Dans les autres cas, on enregistre un *callback* pour le message à recevoir, puis on laisse la main pour que d'autres domaines réactifs puissent s'exécuter entre temps. Le problème est que cela requiert d'avoir accès au reste du calcul sous forme de continuation, ce qui n'est pas possible avec l'architecture actuelle du code pour le cas de `Msignal` sans complexifier énormément le code. On touche ici aux limites de l'approche événementielle de la concurrence. On pourrait se demander si l'on ne pourrait pas résoudre ce problème en voyant le moteur d'exécution distribué comme un programme REACTIVEML et en laissant le compilateur REACTIVEML gérer cette transformation en continuation.

Remarque 15. De la même façon, on n'a pas accès à la continuation du programme pour la création d'un signal au sein d'une expression instantanée, puisque le compilateur n'effectue pas de transformation CPS sur les parties ML du code. On décide donc de restreindre la création de signaux aux processus en modifiant l'analyse de bonne formation de la partie 6.2.

Résultats expérimentaux

Nous suivons la même procédure que pour les tests du moteur d'exécution parallèle (voir page 160). On lance les processus sur la même machine et ils communiquent par mémoire partagée. On utilise cette fois MAC OS X 10.6.8 et OCAML 4.0.1. La référence est la version OCAML séquentielle compilée en code natif.

Comme pour le moteur d'exécution parallèle en mémoire partagée, nous commençons par tester les performances en fonction du rapport calcul/synchronisation. On reprend donc le processus `test_compute` (page 160), où l'on ajoute un domaine réactif pour le corps du processus :

```
let process test_compute_domain  =
  for i=1 to n dopar
    domain ck by 1 do
      loop do_stuff compute_intensity; pause global_ck end
    done
  done
```

Dans le cas d'un processus ne dépendant pas de l'horloge locale et ne faisant pas de dépendance immédiate, le fait de le lancer à l'intérieur d'un domaine de période 1 ne change pas sa sémantique. Dans cet exemple, cela permet aux différentes instances de `do_stuff` de s'exécuter en parallèle, puisque l'on ne parallélise que les domaines réactifs entre eux. On peut observer le résultat sur le premier graphe de la figure 9.5. On remarque que l'on obtient une bonne accélération lorsqu'il y a beaucoup de calcul, proche du moteur d'exécution en mémoire partagée, mais qu'il y a une forte décélération lorsque le calcul devient minoritaire face aux communications. L'exécution du programme devient ainsi cinquante fois plus lente lorsque l'on ne calcule pas et que l'on ne fait que se synchroniser.

On retrouve les mêmes conclusions sur les exemples, que l'on présente sur le second graphe. On peut obtenir une accélération, mais il faut pour cela que l'on calcule beaucoup entre chaque synchronisation. Dans l'exemple des n-corps, il faut quadrupler le nombre de planètes pour obtenir un vrai gain dans l'exécution parallèle. Celui-ci reste limité, puisque la version parallèle n'est que 60% plus rapide.

Le graphe présente également les résultats en ne lançant qu'un seul processus, ce qui permet de tester le coût induit par le moteur d'exécution basé sur l'envoi de messages. Celui-ci est faible dans la plupart des cas (2 % pour la simulation des n-corps), mais il peut devenir très important, comme dans l'exemple de la simulation de réseau de capteurs de la figure 3.2 (page 43) où chaque domaine exécute mille instants locaux. On voit ainsi sur le graphe que cette version du moteur d'exécution est six fois plus lente si l'on utilise un seul processus. Le gain lié à l'exécution parallèle ne permet pas de combler ce retard.

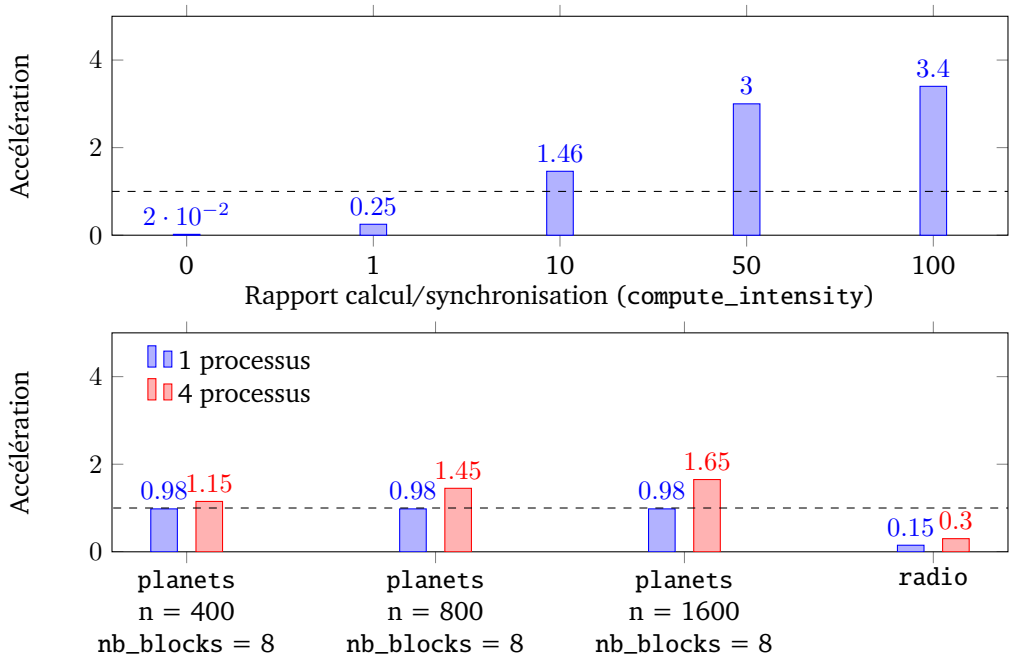


FIGURE 9.5 – Résultats expérimentaux du moteur d'exécution avec envoi de messages

CHAPITRE 10

Interprétation dynamique d'Esterel

Dans ce chapitre, nous allons mettre en pratique les domaines réactifs et les autres nouveautés du langage sur un exemple un peu plus conséquent. Il s'agit d'un interprète dynamique du langage ESTEREL. Dans un second temps, nous discuterons de la possibilité d'ajouter les domaines réactifs en ESTEREL et nous verrons que la causalité constructive d'ESTEREL [Ber96] constitue encore la bonne notion de programmes « raisonnables ».

10.1 Un interprète ESTEREL en REACTIVEML avec domaines réactifs

Nous commençons par rappeler les principes d'ESTEREL et de sa sémantique constructive. Nous décrirons ensuite notre interprète, d'abord du langage sans exceptions puis du langage complet.

La sémantique constructive d'ESTEREL

Le langage ESTEREL ESTEREL [Ber97] est le premier langage synchrone impératif, dont REACTIVEC et REACTIVEML s'inspirent. Comme dans REACTIVEML, on retrouve les notions de séquence, de composition parallèle synchrone, de boucles, etc. Nous considérons dans ce chapitre le noyau du langage tiré de [Ber96] :

$$
\begin{aligned}
e ::= {} & \texttt{nothing} \mid \texttt{pause} \mid e; e \mid e \parallel e \mid \texttt{loop}\ e\ \texttt{end} \\
& \mid \texttt{signal}\ s\ \texttt{in}\ e\ \texttt{end} \mid \texttt{emit}\ s \mid \texttt{present}\ s\ \texttt{then}\ e\ \texttt{else}\ e\ \texttt{end} \\
& \mid \texttt{trap}\ T\ \texttt{in}\ e\ \texttt{end} \mid \texttt{exit}\ T
\end{aligned}
$$

L'expression `nothing` termine instantanément, alors que `pause` attend l'instant suivant. On dispose également de la séquence $e; e$, la composition parallèle $e \parallel e$ et la boucle inconditionnelle `loop` e `end`. La déclaration d'un signal local s est notée `signal` s `in` e `end`, l'émission sur un signal est notée `emit` s et le test de présence `present` s `then` e `else` e `end`. Enfin, le langage permet également de déclarer une exception T avec la syntaxe `trap` T `in` e `end` et de lancer cette exception avec `exit` T.

Voici un exemple du programme ESTEREL le plus célèbre, tiré de [Ber97], qui émet sa sortie `O` une fois qu'il a reçu ses deux entrées `A` et `B`. La troisième entrée `R` permet de remettre le processus dans son état initial :

```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```

Au delà des détails de syntaxe, il existe une différence profonde entre REACTIVEML et ESTEREL. En ESTEREL, on peut réagir instantanément à la présence *et* à l'absence d'un signal. Comme on l'a déjà vu dans l'introduction, cela peut aboutir à des programmes contradictoires, dans lesquels un signal est à la fois absent et présent au cours d'un même instant. On obtient alors un programme qui n'a pas de sémantique, comme l'exemple suivant :

```
signal S in
  present S then nothing else emit S end
end
```

On peut également écrire des programmes non-déterministes, c'est-à-dire pour lesquels on peut supposer que S est présent ou bien absent sans avoir de contradiction :

```
signal S in
  present S then emit S else nothing end
end
```

Dans le contexte de la programmation de systèmes réactifs critiques, il faut rejeter ces deux programmes et n'accepter que les programmes pour lesquels on peut donner un et un seul statut à chaque signal. C'est ce qu'on appelle des programmes *logiquement corrects* [Ber96] (on dira aussi logiquement causaux). Le choix fait par REACTIVEC et REACTIVEML est de n'autoriser la réaction à l'absence d'un signal qu'avec un instant de retard, ce qui permet de rendre tous les programmes causaux « par construction ». Dans le cas d'ESTEREL, on doit trouver un critère pour distinguer les programmes causaux. On veut en fait se restreindre à une classe plus petite que celle des programmes logiquement corrects. Considérons le programme suivant :

```
signal S in
  present S then emit T else nothing end; emit S end
end
```

Ce programme est logiquement causal car on peut donner un et seul statut au signal S, c'est-à-dire présent, mais on veut quand même le rejeter. En effet, il ne suit pas l'idée que la cause, c'est-à-dire l'émission du signal, précède toujours l'effet, ici la réaction à sa présence. Le but de la sémantique *constructive* d'ESTEREL [Ber96] et de l'analyse de causalité qui l'accompagne est de n'accepter que les programmes logiquement corrects et raisonnables du point de vue de la causalité.

La sémantique constructive Le principe de la sémantique constructive est que l'exécution d'un programme ne fait que propager les faits connus sur la présence et l'absence des signaux sans jamais faire d'hypothèses. Nous en rappelons ici les grandes lignes. On trouvera une description formelle dans [Ber96]. On utilise un environnement de signaux E qui donne le statut des signaux, c'est-à-dire présent, absent ou inconnu. Étant donné un programme p et un environnement de signaux E, on définit deux prédicats :

- $\mathrm{Must}(p, E)$ renvoie la liste des signaux nécessairement émis par p.
- $\mathrm{Can}(p, E)$ renvoie la liste des signaux potentiellement émis par p.

Un signal est présent s'il appartient à l'ensemble $\mathrm{Must}(p)$, c'est-à-dire qu'il est nécessairement émis, et il est absent s'il n'appartient pas à $\mathrm{Can}(p)$, c'est-à-dire s'il ne peut pas être émis. On calcule ces deux fonctions par un point fixe :

1. On parcourt d'abord le programme pour connaître les signaux nécessairement émis. On change alors leur statut dans l'environnement de signaux E pour enregistrer leur présence. En particulier, dans le cas de `present S then p1 else p2 end`, on parcourt `p1` si on sait que S est présent ou `p2` si S est absent. On ne parcourt aucune branche si le statut de S est inconnu.
2. En utilisant les informations de la phase précédente, on calcule l'ensemble des signaux qui peuvent être potentiellement émis. On donne alors le statut 'absent' aux signaux qui ne peuvent pas être émis. En particulier, dans le cas de `present S then p1 else p2 end`, on parcourt `p1` si on sait que S est présent ou `p2` si S est absent. On parcourt les deux branches pour les signaux dont on ne connait pas encore le statut.

3. Si l'environnement de signaux E contient encore des signaux avec un statut inconnu, on recommence à la première étape jusqu'à avoir déterminé le statut de tous les signaux.

Les programmes pour lesquels le point fixe n'est jamais atteint sont considérés comme incorrects et n'ont pas de sémantique. L'analyse de causalité d'ESTEREL permet de s'assurer statiquement, au moment de la compilation, qu'un programme est causal, c'est-à-dire que l'algorithme ci-dessus va converger vers un point fixe unique. Elle s'appuie sur la traduction en circuit du langage et est définie formellement dans [SBT96].

Nous allons illustrer la sémantique constructive avec le programme suivant :

```
present S1 then nothing else emit S2 end
||
present S2 then emit O else nothing end
||
present S3 then nothing else emit S1 end
```

On commence par calculer l'ensemble des signaux nécessairement émis dans un environnement de signaux E_0 dans lequel le statut de tous les signaux est inconnu. L'ensemble $\mathrm{Must}(p, E_0)$ est vide et $\mathrm{Can}(p, E_0) = \{\mathtt{S1}, \mathtt{S2}, \mathtt{O}\}$. Cela signifie que S1, S2 et O peuvent potentiellement être émis, mais pas S3. On peut donc en déduire que S3 est nécessairement absent et que la branche **else** sera exécutée. On obtient alors un nouvel environnement E_1, dans lequel S3 est absent et le statut des autres signaux reste inconnu. On a alors $\mathrm{Must}(p, E_1) = \{\mathtt{S1}\}$. On donne donc le statut 'présent' à S1 et on obtient un nouvel environnement de signaux E_2. On a alors $\mathrm{Can}(p, E_2) = \{\mathtt{S1}, \mathtt{O}\}$. S2 n'apparait plus dans la liste des signaux potentiellement émis et il est donc absent. On en déduit finalement que O est également absent. Au final, on obtient donc que S1 est présent, alors que S2, S3 et O sont absents.

Interprétation sans exceptions

Motivation Alors que REACTIVEML utilise un moteur d'exécution pour obtenir un ordonnancement dynamique, la concurrence d'ESTEREL est compilée vers du code séquentiel avec ordonnancement statique, après une analyse de causalité préalable pour rejeter les programmes incorrects. Les premières versions du langage compilent les programmes ESTEREL vers des automates [BG92], dont la taille est exponentielle en la taille du programme source. La traduction d'ESTEREL en circuits [Ber92] utilisée dans les versions suivantes permet d'éviter l'explosion combinatoire des automates et de générer aussi bien un programme qu'un circuit à partir du même programme ESTEREL. D'autres approches [CPP+02, Edw00] se spécialisent dans la génération de code séquentiel et offrent de meilleures performances en logiciel. Plus récemment, le langage HIPHOP [BNS11] intègre ESTEREL dans HOP [1], qui est un langage *multi-tier* pour la programmation de services Web. Son moteur d'exécution traduit les constructions ESTEREL en un arbre de syntaxe abstraite (AST) qui est ensuite interprété dynamiquement suivant la sémantique constructive de [Ber96].

Nous cherchons dans ce chapitre à réaliser un interprète dynamique d'ESTEREL, sans aucune analyse préalable et le plus efficace possible. Le but de cette expérience est de savoir si on peut encoder la dépendance instantanée à l'absence d'un signal d'ESTEREL en utilisant le retard à l'absence et les domaines réactifs de REACTIVEML. Nous allons pour cela itérer l'exécution des programmes comme dans la sémantique constructive d'ESTEREL, puis utiliser les domaines réactifs pour masquer ces instants de calcul. Par rapport à HIPHOP, nous cherchons à réutiliser au maximum les constructions de REACTIVEML, comme le parallélisme synchrone et les signaux, plutôt que d'écrire un interprète manipulant un AST. Cette expérience nous permettra aussi de valider les nouveautés que nous avons présentées dans ce manuscrit sur un exemple conséquent.

Principe Notre interprète ESTEREL va suivre le principe de la sémantique constructive et itérer l'exécution du programme pour propager les statuts des signaux. Il alterne donc les phases d'exécution, correspondant au calcul de Must, avec des phases de prédiction, correspondant au calcul

1. http://hop.inria.fr/

de Can. On utilise un domaine réactif pour masquer les instants locaux permettant de calculer les statuts des signaux. On obtient alors un comportement quasiment équivalent au programme ESTEREL, c'est-à-dire que chaque instant du programme ESTEREL correspond à un instant de l'interprète. Plus précisément, si on considère notre interprète comme une boite noire et que les entrées et sorties du programme ESTEREL sont des signaux REACTIVEML dans l'interprète, alors les sorties seront décalées d'un instant. En effet, la propagation du statut des signaux ESTEREL se fait instantanément du point de vue de l'horloge globale de l'interprète. Mais la lecture des entrées du programme ESTEREL se fait avec un instant de retard, puisque le corps de l'interprète, qui est à l'intérieur d'un domaine réactif, ne peut dépendre instantanément de l'absence ou la présence des entrées du programme. Le code de l'interprète lui-même fait environ 300 lignes de REACTIVEML, auxquelles il faut rajouter quelques centaines de lignes pour l'analyse syntaxique du noyau d'ESTEREL considéré et les fonctions auxiliaires.

Le principe de l'interprète est de traduire le programme ESTEREL de façon structurelle, en associant à chaque expression un combinateur de la forme :

```
let process pexp env (step_in, predict_in) (step_out, predict_out) = ...
```

L'environnement `env` contient une table associant un signal à un identifiant. Les autres entrées sont des signaux, deux entrées et deux sorties, qui permettent de gérer l'exécution de l'expression. Le signal `step_in` indique le début de l'exécution de l'expression et l'expression émet `step_out` pour signifier la fin de son exécution. Ces signaux sont émis une seule fois pour chaque expression. Les signaux `predict_in` et `predict_out` jouent le même rôle pour la phase de prédiction, mais ils peuvent être émis plusieurs fois. En outre, les phases de prédiction précèdent les phases d'exécution, c'est-à-dire que le signal `predict_in` ne peut pas être émis après `step_in` (mais ils peuvent être émis dans le même instant). On assure également que tous les combinateurs terminent leur exécution après l'émission de `step_out`. Le lancement d'un programme est fait par le processus suivant (le symbole **@** indique que l'on peut cliquer sur le nom du fichier pour accéder au code complet) :

@ exemples/EsteRML/combinators.rml

```
let process run_program env p =
  signal step_in, predict_in, step_out, predict_out in
  run p env (step_in, predict_in) (step_out, predict_out)
  ||
  (* execution *)
  emit step_in; emit predict_in
```

Il suffit d'émettre les signaux `step_in` et `predict_in` une seule fois. Les ordres d'exécution et de prédiction seront ensuite transmis au reste de l'expression au cours de son exécution. On termine le programme une fois que l'expression a terminé son exécution, c'est-à-dire lorsqu'elle émet `step_out`.

Séquence La gestion de la séquence est immédiate avec le choix de la forme des combinateurs. Il suffit de connecter les signaux de sortie de la première expression `p1` avec les signaux d'entrée de la seconde `p2`. On obtient ainsi le combinateur `pseq` suivant :

```
let process pseq p1 p2 env
     (step_in, predict_in) (step_out, predict_out) =
  signal l_step, l_predict in
  run p1 env (step_in, predict_in) (l_step, l_predict)
  ||
  run p2 env (l_step, l_predict) (step_out, predict_out)
```

On peut remarquer que l'on lance en parallèle les combinateurs correspondant aux deux expressions puisque l'on utilise les signaux `step_in` et `step_out` pour gérer l'ordre d'exécution et pas la terminaison des combinateurs.

Expressions de base Le cas le plus simple est celui de l'expression **nothing** qui termine instantanément et que l'on traduit en un combinateur pnothing défini par :

```
let process pnothing env
    (step_in, predict_in) (step_out, predict_out) =
  (* execution *)
  await immediate step_in;
  emit step_out
  ||

  (* prediction *)
  do
    loop
      await immediate predict_in;
      emit predict_out;
      quiet pause
    end
  until step_out done
```

Le combinateur émet step_out dès la réception de step_in et transmet également toutes les activations de predict_in. On termine l'exécution une fois le signal step_out émis. On utilise dans la boucle l'instruction **quiet pause** car on va également utiliser un domaine réactif pour décider de la fin de l'instant du programme ESTEREL. On rappelle que quiet **pause** ck attend le prochain instant de ck mais doit être considéré en attente par le domaine réactif et ne pas influencer le choix de faire un autre instant local. Les processus annexes qui transmettent les signaux utilisent donc tous **quiet pause** pour ne pas influencer le calcul de la fin de l'instant. La fin de l'instant ESTEREL se fait lorsque tous les processus appellent **pause**, que l'on traduit par l'appel de **pause** global_ck :

```
let process ppause env (step_in, predict_in) (step_out, predict_out) =
  (* execution *)
  await immediate step_in;
  pause global_ck;
  emit step_out;
  emit predict_out
```

On peut remarquer que l'on lance les phases de prédiction et d'exécution à l'instant suivant de l'horloge globale, qui correspond à l'instant suivant du programme ESTEREL.

Composition parallèle La composition parallèle synchrone nécessite un peu plus de travail :

```
let process ppar p1 p2 env (step_in, predict_in) (step_out, predict_out) =
  signal s_out1, s_out2, p_out1, p_out2 in
     run p1 env (step_in, predict_in) (s_out1, p_out1)
  || run p2 env (step_in, predict_in) (s_out2, p_out2)
  || (* execution *)
  do
    (await immediate s_out1 || await immediate s_out2); emit step_out
    || (* prediction *)
    loop
      await immediate (p_out1 /\ p_out2); emit predict_out; quiet pause
    end
    || (* maintient p_out1 tant que la seconde branche n'a pas termine *)
    await immediate s_out1; run hold p_out1
    || (* idem pour p_out2 *)
    await immediate s_out2; run hold p_out2
  until step_out done
```

Pour l'exécution, on attend la fin des deux branches avant d'émettre `step_out`. On utilise pour cela une *configuration événementielle* : `await immediate (p_out1 /\ p_out2)` termine instantanément lorsque les signaux `p_out1` et `p_out2` sont émis pendant le même instant. On lance la prédiction sur la suite du code si les deux branches peuvent potentiellement terminer dans l'instant, c'est-à-dire si leurs signaux `predict_out` sont émis. Il faut également penser à maintenir le signal de prédiction sortant d'une branche en attendant que l'autre termine.

Signaux On traduit chaque signal en une structure de données contenant un signal à mémoire `r_status` donnant le statut du signal (`Present`, `Absent` ou `Unknown`) et un signal `r_may_emit` qui sera émis lors des phases de prédiction pour indiquer que le signal est potentiellement émis. On utilise la réinitialisation d'un signal (avec le mot-clé **reset**) pour mettre le statut de chaque signal à `Unknown` au début de chaque instant de l'horloge globale, qui correspond à un instant du programme ESTEREL. L'émission d'un signal se traduit par :

```
let process pemit s env (step_in, predict_in) (step_out, predict_out) =
  let rev = event_of_id env s in
  (* execution *)
  await immediate step_in;
  rev.r_status <== Present;
  emit step_out
  ||
  (* prediction *)
  do
    loop
      await immediate predict_in;
      emit rev.r_may_emit;
      emit predict_out;
      quiet pause
    end
  until step_out done
```

Lors de la phase d'exécution, on change le statut du signal en `Present`, alors que l'on émet le signal `r_may_emit` lors de la phase de prédiction.

Test de présence Le cœur de l'interprète est la traduction de l'opérateur **present**, que l'on peut voir sur la figure 10.1. On commence par tester la présence du signal (ligne 10). S'il est présent ou absent, alors on lance la branche correspondante. Si le statut du signal est inconnu, on lance la prédiction dans les deux branches en parallèle (ligne 14). Si le signal `r_may_emit` est absent, alors le signal ne peut pas être émis et on peut donc en déduire qu'il est absent (ligne 18). Enfin, pour les phases de prédiction, il est important de prendre en compte le statut du signal et de ne parcourir que les branches potentiellement exécutables. Il ne s'agit pas d'une optimisation, mais bien d'une des composantes de la sémantique constructive.

Boucles Enfin, la gestion des boucles utilise la création dynamique :

```
let process ploop p env (step_in, predict_in) _ =
  let rec process aux (s_in, p_in) =
    signal p_out, s_out in
    run p env (s_in, p_in) (s_out, p_out)
    ||
    (pause global_ck || await immediate (s_out \/ p_out));
    run aux (s_out, p_out)
  in
  run aux (step_in, predict_in)
```

```
let process ppresent s p1 p2 env
      (step_in, predict_in) (step_out, predict_out) =
  let rev = event_of_id env s in
  signal s_in1, s_in2, p_in1, p_in2 in
  run p1 env (s_in1, p_in1) (step_out, predict_out)
  ||
  run p2 env (s_in2, p_in2) (step_out, predict_out)
  ||
  do
    begin
      let rec process check_status =
        match last rev.r_status with
        | Present -> emit s_in1
        | Absent -> emit s_in2
        | Unknown ->
          emit p_in1; emit p_in2;
          present rev.r_may_emit
          then pause
          else (if last rev.r_status = Unknown
                then (rev.r_status <== Absent));
          run check_status
      in
      await immediate step_in;
      run check_status
    end
    ||
    (* prediction *)
    loop
      await immediate predict_in;
      (match last rev.r_status with
      | Present -> emit p_in1
      | Absent -> emit p_in2
      | Unknown -> emit p_in1; emit p_in2);
      quiet pause
    end
  until step_out done
```

FIGURE 10.1 – Traduction de l'opérateur `present`

On lance d'abord une première instance du corps de la boucle, avec des nouveaux signaux `s_out` et `p_out` pour la sortie de l'exécution et de la prédiction. Lorsqu'un de ces signaux est émis, on crée de façon paresseuse une nouvelle instance du corps de la boucle, avec des nouveaux signaux de sortie et prédiction. Pour éviter de créer un nombre infini de processus, il faut attendre au moins un instant avant de créer une nouvelle instance de la boucle. Cela ne pose pas de soucis puisque, dans un programme ESTEREL bien formé (on dit aussi *loop safe* dans [Ber96]), le corps des boucles n'est jamais instantané. Cet encodage des boucles évite les problèmes de réincarnation [TdS05], puisque l'on créé une nouvelle instance du processus `p` pour chaque itération de la boucle.

Ajout des exceptions

Les exceptions en ESTEREL Une des autres particularités d'ESTEREL par rapport à REACTIVEML est le mécanisme d'exception. Il s'agit d'un mécanisme de préemption, mais qui présente plusieurs différences avec la construction **`do`**/**`until`**. On définit cette structure de contrôle avec le mot-clé **`trap`** suivi du nom de l'exception et on appelle **`exit`** pour lancer l'exception. On peut écrire par exemple :

```
trap T in
     emit A
  || exit T; emit B
  || pause; emit C
end;
emit D
```

Puisque l'on lance l'exception `T`, le corps est préempté à la fin du premier instant, donc le signal `C` n'est jamais émis. Il s'agit d'une préemption faible, donc on exécute bien les autres branches et on émet le signal `A`. La construction **`exit`** ne termine jamais, donc `B` n'est pas émis. Enfin, la construction **`trap`** termine instantanément lorsque l'exception est levée, contrairement au **`do`**/**`until`**. Le signal `D` est donc aussi émis au cours du premier instant. L'autre subtilité concerne les **`trap`** imbriqués :

```
trap T in
  trap U in
    exit T || exit U
  end;
  emit A
end;
emit B
```

Dans ce cas, on exécute seulement la continuation de l'exception la plus englobante qui est levée. Dans l'exemple, seul `B` est émis. Pour obtenir ce résultat, chaque expression renvoie un code de retour :

- 0 signifie que l'expression termine instantanément.
- 1 signifie que l'expression attend l'instant suivant.
- $2 + i$ signifie que l'exception i a été levée. On utilise ici des indices de De Bruijn. Le cas $i = 0$ représente l'exception la moins englobante, $i = 1$ la suivante, et ainsi de suite.

Ce choix permet de gérer facilement la composition parallèle, puisqu'il suffit de renvoyer le maximum des codes de retour des deux branches. Cela permet de gérer à la fois le fait que le parallèle termine lorsque les deux branches ont terminé et la priorité des exceptions. On exécute la continuation d'un **`trap`** si le corps renvoie 0 (c'est-à-dire qu'il termine) ou 2 (l'exception a été levée et aucune exception plus englobante n'a été levée). Les détails de la gestion des exceptions sont disponibles dans [Ber96].

Implémentation On ajoute un argument `must_k` en entrée des combinateurs, qui est un signal sur lequel l'expression émet son statut de retour. Le code de retour 0 est déjà représenté par les signaux `step_in` et `step_out`, donc on n'émet que les codes supérieurs, pour l'opérateur **pause** et les exceptions. La composition parallèle collecte les valeurs renvoyées par les deux branches et renvoie leur maximum.

Il faut également gérer le cas des phases de prédiction. On ajoute donc un autre signal `can_k` sur lequel on émet les codes de retour potentiels de l'expression. On lance la prédiction sur la suite d'un **trap** si son corps peut potentiellement renvoyer le code 0 ou 2. Pour collecter les différents codes de retour, on utilise naturellement la multi-émission. Mais le problème est que la prédiction peut prendre plusieurs instants, puisque la composition parallèle doit d'abord collecter les valeurs des deux branches avant de pouvoir émettre ses codes de retour potentiels. Il faut donc accumuler les valeurs sur plusieurs instants. Il se pose également le problème de savoir quand on a reçu toutes les valeurs possibles. On va là encore utiliser astucieusement un domaine réactif, dans lequel on lance les combinateurs associés aux deux branches du parallèle. En déclarant les signaux pour les codes de retour en dehors du domaine, on collecte les valeurs sur plusieurs instants. On utilise également le domaine réactif pour décider de la fin de la prédiction : elle est terminée lorsque tous les processus à l'intérieur du domaine, c'est-à-dire le corps des branches, ont terminé la phase de prédiction et attendent l'instant suivant de l'horloge de prédiction `env.m_eval_ck`. On procède de la même façon pour la construction **trap** :

@ exemples/EsteRML/combinators_code.rml

```
let process ptrap k p env (must_k, can_k)
          (step_in, predict_in) (step_out, predict_out) =
  signal kill clock global_ck in
  signal l_must_k, l_can_k in
  let env = new_trap env k in
  domain ck do
    do
     run p env (l_must_k, l_can_k) (step_in, predict_in) (step_out, predict_out)
    until kill done
  done
  ||
  do (* execution *)
    await immediate one l_must_k(c) in
    if c = 2
    then (emit kill; emit step_out)
    else emit must_k (minus_trap c)
    ||
    (* prediction *)
    loop
      await l_can_k (vl) in
      if List.mem 2 vl then emit predict_out;
      quiet pause env.m_eval_ck
    end
  until step_out done
```

On lance le corps `p` à l'intérieur d'une préemption sur le signal `kill` pour pouvoir terminer son exécution lorsque l'exception `k` est levée. Comme il s'agit de préemption faible, l'horloge du signal `kill` est l'horloge globale, puisque l'on doit finir l'exécution de l'instant courant du corps du **trap**. Cette préemption est elle-même à l'intérieur d'un domaine réactif pour gérer la prédiction du code de retour. Du point de vue de l'exécution, on attend le code de retour du corps du **trap**. Si celui-ci est égal à 2, alors on effectue la préemption et on exécute la continuation du **trap** en émettant le signal `step_out`. Sinon, on transmet le code de retour en le décrémentant avec la fonction `minus_trap` (puisqu'il s'agit d'indices de De Bruijn). Pour la prédiction, on lance la prédiction sur la continuation si le corps du **trap** peut terminer avec le code de retour 2.

Une nouvelle construction Le fait que la prédiction prenne plusieurs instants pose un autre problème. On a vu dans le cas de la composition parallèle que l'on attendait le signal de prédiction des deux branches en appelant :

```
await immediate (p_out1 /\ p_out2)
```

Puisque les signaux peuvent maintenant être émis pendant des instants locaux différents, on souhaite remplacer cette construction par un combinateur `await_both_reset` qui attend que les signaux `a` et `b` soient émis au cours du même instant de l'horloge `rck` (qui est plus lente que l'horloge de `a` et `b`). On peut écrire le processus suivant :

```
let process await_both_reset a b rck =
  signal kill in
  do
    loop
      signal s clock rck in
      do
        (await immediate a || await immediate b);
        emit kill; quiet pause
      until s done
      ||
      emit s
    end
  until kill done
```

Lorsque l'on a reçu les deux signaux, on émet le signal `kill` (qui est lui un signal rapide) pour que le processus complet termine. On doit attendre l'instant suivant en appelant **`quiet pause`** après l'émission de `kill` pour ne pas que le corps de la boucle termine instantanément et éviter ainsi de recommencer une nouvelle instance de la boucle. Si on n'a pas reçu un des deux signaux, on préempte le corps à la fin de l'instant de `rck` et on recommence dans l'état initial. Le problème est que ce processus rend le domaine d'horloge `rck` non coopératif si jamais les signaux ne sont pas présents en même temps. En effet, à la fin de chaque instant de `rck`, le corps du processus est préempté et on exécute sa continuation à l'instant suivant de `rck`. On recommence ensuite une nouvelle instance de la boucle, et ainsi de suite. Formellement, on peut voir dans la définition du prédicat $\vdash_{\text{next}}$ (règle NEXTUNTIL de la figure 5.1) qu'une préemption doit toujours exécuter un autre instant si le signal est présent.

Pour résoudre ce problème, on introduit comme pour la construction **`pause`** une variante de la structure de contrôle **`do/until`**. Sa continuation s'exécute à l'instant suivant de l'horloge du signal de préemption, mais elle ne demande pas l'exécution d'un autre instant. On note **`do e quiet until s done`** cette nouvelle construction. La sémantique de cette opération suit le même principe que la construction **`quiet pause`**. Son comportement est le même que la préemption **`do e until s done`**, mais elle ne vérifie pas la règle NEXTUNTIL. On peut de la même façon introduire des variantes des constructions **`present`** et **`await`**. On corrige donc l'exemple en écrivant :

```
...
do
  (await immediate a || await immediate b);
  emit kill; quiet pause
quiet until s done
...
```

Discussion

Détection des erreurs de causalité Lorsqu'un programme n'est pas constructivement correct, l'alternance entre phases d'exécution et de prédiction ne parvient jamais à déterminer le statut de tous les signaux. On peut détecter cette situation en vérifiant à chaque phase de prédiction que l'on a pu décider de l'absence d'au moins un signal. On utilise pour cela le signal `env.m_move`, que l'on doit émettre lorsque l'on décide de l'absence d'un signal (ligne 18 de la figure 10.1). S'il est absent, alors on en déduit à l'exécution que le programme n'est pas constructivement causal. On pourra voir les détails de l'implémentation dans le fichier @ `exemples/EsteRML/combinators.rml`.

Création dynamique L'interprète que nous venons de présenter peut exécuter des programmes ESTEREL avec création dynamique. En effet, le calcul du statut des signaux se fait de façon dynamique, sans jamais connaître le nombre d'émetteurs potentiels d'un signal.

On commence par ajouter la notion de processus, en stockant dans l'environnement une table qui associe un processus à un nom. On peut y ajouter un processus avec la fonction `new_process` et y accéder avec `process_of_id`. Le combinateur `prun` permet de lancer un tel processus, en allant le chercher dans l'environnement :

```
let process prun f env (step_in, predict_in) (step_out, predict_out) =
  let p = process_of_id env f in
  run p env (step_in, predict_in) (step_out, predict_out)
```

On peut ensuite ajouter un combinateur `prec` pour créer un processus récursif :

```
let process prec f p env (step_in, predict_in) (step_out, predict_out) =
  let process self env (step_in, predict_in) (step_out, predict_out) =
    (pause global_ck || await immediate (step_in \/ predict_in));
    run p env (step_in, predict_in) (step_out, predict_out)
  in
  let env = new_process env f self in
  run p env (step_in, predict_in) (step_out, predict_out)
```

On réutilise le même principe de création paresseuse que pour l'encodage de la boucle afin de ne pas créer un nombre infini de processus. On ajoute dans la table un processus qui attend l'émission d'un des signaux `step_in` ou `predict_in` pour créer une nouvelle instance du processus. On peut noter au passage que si l'on combine les combinateurs correspondant à la récursion et à la séquence, on obtient bien un code équivalent à celui de la boucle. La création dynamique n'est pas vraiment utile dans notre interprète puisque le langage ne contient pas de valeurs, mais cette petite expérience montre qu'elle est possible.

Programmation avec les domaines réactifs On peut se demander ce que l'on a gagné en utilisant les domaines réactifs pour la programmation de l'interprète ESTEREL. La première réponse est que cela permet d'obtenir un interprète dans lequel un instant de l'interprète correspond à un instant du programme, mais il s'agit finalement d'un détail. Une façon concrète de répondre à cette question est d'écrire le même programme sans domaines réactifs. On ne s'intéresse qu'au cas d'ESTEREL sans exceptions qui est plus simple, mais on peut procéder de la même façon pour le langage complet.

On reprend exactement la même structure et le même fonctionnement. Pour remplacer le domaine réactif, il nous faut tout d'abord déterminer la fin de l'exécution d'un instant du programme ESTEREL et synchroniser les différents processus. On définit pour cela un signal `m_active` émis tant que le calcul est en cours, un signal `m_step` indiquant le début de l'exécution d'un instant et enfin un signal `m_step_done` signalant la fin de l'instant qui est utilisé pour remettre à zéro les statuts des signaux. On retrouve à nouveau les principes que l'on a déjà décrits à la fin du chapitre 2. On programme ensuite un ordonnanceur chargé de coordonner l'exécution :

@ exemples/EsteRML/combinators_nodo.rml

```
let process run_program =
  ... ||
  let rec process check_move =
    present env.m_active then (pause; run check_move)
  in
  do
    loop
      run check_move; emit env.m_step_done;
      pause;
      emit env.m_step;
      pause
    end
  until step_out done
```

On attend que le signal `env.m_active` soit absent avec le processus `check_move`, puis on émet le signal `env.m_step_done` et enfin on recommence un nouvel instant en émettant `env.m_step`. Il faut également modifier le code pour émettre le signal `m_active` lorsque le calcul est en cours, c'est-à-dire un peu partout dans le code. On remplace aussi les appels à `pause global_ck` par l'attente du signal `m_step`. Il faut également faire un peu de travail pour remettre à zéro le statut des signaux à chaque émission du signal `m_step_done` :

```
let process psignal s p env (step_in, predict_in) (step_out, predict_out) =
  let env = run new_event env s in
  run p env (step_in, predict_in) (step_out, predict_out)
  ||
  signal kill in
  do
    let rev = event_of_id env s in
    loop
      await immediate env.m_step_done;
      rev.r_status <== Unknown;
      await immediate env.m_step
    end
  until kill done
  ||
  await immediate step_out; emit kill
```

On lance en parallèle de la création du signal un processus qui met le statut du signal à `Unknown` à la fin de chaque instant du programme ESTEREL, c'est-à-dire à chaque émission du signal `m_step_done`. On termine ce processus lorsque le corps termine par une préemption sur le signal `kill`.

Dans les différents cas, on voit qu'il s'agit finalement de peu de code, mais que celui-ci est plutôt complexe à écrire. Il faut sans cesse compter le nombre d'instants qui s'écoulent pour bien s'assurer de la synchronisation des processus, faire attention si l'on appelle `await` ou `await immediate`, etc.

On peut faire d'autres remarques sur la programmation avec les domaines réactifs :

- En plus de masquer des calculs locaux, on utilise souvent les domaines réactifs pour déterminer automatiquement la fin d'un calcul qui prend plusieurs instants, ou autrement dit pour implémenter une barrière. On lance pour cela tous les processus à l'intérieur d'un même domaine d'horloge `ck`. On appelle `pause ck` pour effectuer un pas de calcul supplémentaire et `pause global_ck` lorsque l'on a terminé et que l'on attend la prochaine phase de calcul. On passe automatiquement à la phase suivante lorsque tous les processus ont terminé la phase courante, grâce à la propriété d'attente automatique des domaines réactifs. On se sert aussi souvent des signaux lents pour combiner la valeur d'un signal sur plusieurs instants de façon transparente.
- Une bonne pratique de programmation avec les domaines réactifs est d'éviter d'utiliser l'horloge locale `local_ck`. Ainsi, il est préférable de ne pas appeler `pause` sans argument et

de toujours donner l'horloge d'un signal avec le mot-clé **clock** au moment de sa définition. Ainsi, le comportement du programme ne change pas si on ajoute ou enlève un domaine réactif.
– L'impossibilité de dépendre instantanément d'un signal lent est souvent gênante. Il faut remplacer les dépendances immédiates (comme **await immediate**) par des dépendances avec retard (comme **await** sans **immediate**). Le calcul peut alors prendre plusieurs instants, mais on peut toujours utiliser un autre domaine pour masquer ces instants locaux.

Domaines et polymorphisme de rang supérieur Dans le cas du combinateur `ptrap` (page 177), on doit lancer le processus `p` pris en argument à l'intérieur d'un domaine réactif défini localement. Ce programme est donc rejeté par le calcul d'horloges, comme on l'a vu par exemple avec le processus `run_domain` (page 108). Pour remédier à ce problème, on doit définir un type enregistrement `rprocess_poly` décrivant un combinateur dont l'horloge d'activation est quantifiée universellement, afin que l'on puisse le lancer dans le domaine local sans avoir de problème d'échappement de portée :

```
type rprocess{''c|'ck, 'sck, 'sck2, 'pck|'ef} =
  state{|'ck, 'pck|}
    -> ((int, int list) event{|'sck|} * (int, int list) event{|'sck|})
    -> ((unit, unit list) event{|'sck|} * (unit, unit list) event{|'sck|})
    -> ((unit, unit list) event{|'sck2|} * (unit, unit list) event{|'sck2|})
    -> unit process{''c||'ef}
and rprocess_poly{|'ck, 'sck, 'sck2, 'pck|'ef} =
    { l : ''c. rprocess{''c|'ck,'sck, 'sck2, 'pck|'ef} }
```

Les différentes variables de type (`'ck, 'sck, 'sck2, 'pck, 'ef`) représentent des inconnues dont on ne se soucie pas. Le point important est que l'horloge d'activation `''c` du combinateur est quantifiée universellement.

Ce changement pose plusieurs problèmes. Le premier est que l'on ne peut plus utiliser d'application partielle pour les combinateurs. En effet, une application est une expression expansive, donc son type ne peut pas être généralisé. En particulier, l'horloge d'activation d'un processus obtenu par une application partielle ne peut pas être quantifiée universellement. Considérons par exemple la fonction `build_program` qui associe à une expression le combinateur correspondant. Dans la version sans exceptions, elle est simplement définie par :

@ exemples/EsteRML/combinators.rml
```
let rec build_program p = match p with
| Pnothing -> pnothing
| Pseq (e1, e2) -> pseq (build_program e1) (build_program e2)
...
```

Puisqu'on ne peut plus utiliser l'application partielle, on doit définir à la place un processus local :

@ exemples/EsteRML/combinators_code.rml
```
let rec build_program p = match p with
| Pnothing -> { l = pnothing }
| Pseq (e1, e2) ->
    let p1 = build_program e1 in
    let p2 = build_program e2 in
    let process p env (must_k, can_k) (step_in, predict_in)
        (step_out, predict_out) =
      run pseq p1 p2 env (must_k, can_k)
        (step_in, predict_in)  (step_out, predict_out)
    in
    { l = p }
...
```

On voit que le code est bien plus complexe et qu'il est nécessaire d'avoir une bonne compréhension de ML et de la généralisation des variables de type pour résoudre ce problème.

Un autre problème est lié aux effets que l'on peut exprimer dans la signature d'un processus, ce que l'on a déjà mentionné dans la partie 8.3. Dans le cas des combinateurs, on peut remarquer dans la définition du type `rprocess` que l'effet du processus est une variable de rangée d'effets. Cela signifie que l'effet d'un combinateur ne peut pas mentionner l'horloge d'activation `''c` du processus. En effet, lors de l'unification d'un schéma de type, les variables quantifiées universellement ne peuvent pas être unifiées avec des variables non quantifiées. On souhaiterait dire que le combinateur a au moins un effet sur l'horloge `''c`, mais cela est impossible dans notre calcul d'horloges. On ne peut que forcer un processus à avoir un effet vide ou bien un effet réduit à une seule horloge. On arrive tout de même à contourner ce problème à l'aide de plusieurs subterfuges que le lecteur pourra voir dans le fichier `exemples/EsterRML/combinators_code.rml`.

Exécution efficace d'ESTEREL L'écriture de ce petit interprète permet de faire plusieurs remarques sur l'efficacité de l'exécution d'ESTEREL. On peut tout d'abord remarquer que la possibilité de réagir instantanément à l'absence d'un signal coûte cher. On doit itérer plusieurs fois les phases de prédiction et d'exécution pour propager l'information.

L'autre point concerne le choix de la sémantique des `trap` imbriqués. Puisqu'on ne doit exécuter que la continuation de l'exception la plus englobante, il faut transmettre le code de retour de chaque expression en remontant l'arbre de syntaxe abstraite. Cela complexifie l'implémentation et en diminue l'efficacité.

Cette expérience montre qu'il est effectivement possible d'écrire un interprète ESTEREL dynamique respectant la sémantique constructive du langage en utilisant les constructions de REACTIVEML. Malgré tout, on obtient un encodage lourd, utilisant de très nombreux signaux. Il est fort probable qu'un interprète plus basique, manipulant un AST suivant les règles de [Ber96], serait plus efficace. Les choix faits par REACTIVEC et REACTIVEML permettent d'obtenir une exécution plus directe, sans aucune itération et donc bien plus efficace. Cela justifie sans doute la perte d'expressivité liée au retard à l'absence qu'imposent ces langages.

10.2 Les domaines réactifs en ESTEREL

Nous allons maintenant nous éloigner du coté pratique de la programmation avec les domaines réactifs pour nous intéresser à nouveau à la théorie des domaines réactifs. La programmation de cet interprète ESTEREL pose en effet la question de la possibilité d'intégrer les domaines réactifs dans ESTEREL en général et dans l'interprète en particulier. La principale question est de savoir si l'on peut définir une notion de causalité permettant d'accepter les dépendances instantanées sur un signal lent, tout en rejetant les programmes pour lesquels cela pose problème. Nous discuterons également de l'interaction entre la dépendance immédiate à l'absence d'un signal et les domaines réactifs. Nous cherchons ici à donner des intuitions plus qu'une approche formelle, qui constitue une piste de recherche.

Intuition

La réponse à ces questions a déjà été apportée en partie par le *raffinement d'horloges* [GBS10b] dans le langage QUARTZ. Contrairement au choix que nous avons fait dans ce manuscrit, il est possible de dépendre instantanément d'un signal lent. Cela requiert toutefois d'être dans un monde statique où l'on connait tous les émetteurs potentiels de chaque signal. On peut donc donner un sens au programme suivant :

```
signal S in
  domain ck do
    present S then emit O1 else nothing end
    ||
    pause ck; emit O2
  done
  ||
  emit S
end
```

Pendant l'exécution du domaine réactif d'horloge ck, on doit attendre de connaître le statut du signal S pour pouvoir terminer l'exécution du premier instant. Tant que le statut du signal est inconnu, le domaine réactif est bloqué. Une fois que l'on exécute la seconde branche du parallèle et que l'on émet S, on peut terminer le premier instant de l'horloge ck, puis exécuter le second instant. Contrairement au cas de REACTIVEML, l'exécution d'un domaine réactif n'est plus indépendante des autres processus. Elle peut également dépendre de celle d'un autre domaine, comme dans cet exemple :

```
signal S in
  domain ck1 do
    present S then emit O else nothing end
  done
  ||
  domain ck2 do
    pause ck2; emit S
  done
end
```

On ne peut pas exécuter le premier instant de l'horloge ck1 tant que le statut du signal S n'est pas connu. Par contre, on peut effectuer deux instants de l'horloge ck2. On peut donc en déduire la présence de S et continuer l'exécution de ck1. On remarque au passage que ces dépendances entre domaines rendent l'implémentation plus complexe et empêcheraient une exécution parallèle efficace.

Bien entendu, le gain d'expressivité lié à la possibilité de dépendre instantanément d'un signal lent s'accompagne de problèmes de causalité. Nous les avons déjà mentionné avec le processus immediate_dep_wrong (page 36), dont se rapproche le programme suivant :

```
signal S in
  domain ck do
    present S then print "OK" end
    ||
    pause ck; emit S
  done
end
```

Ce programme est *logiquement correct*, c'est-à-dire qu'on est capable de donner un statut unique à tous les signaux sans obtenir de contradiction. En effet, on peut supposer que le signal lent S est présent au cours du premier instant de son horloge. Il est effectivement émis au cours du second instant de l'horloge ck, qui est inclus dans le premier instant de l'horloge de S. Mais, comme dans le cas de la sémantique constructive d'ESTEREL, on souhaite rejeter ce programme car l'effet, c'est-à-dire la réaction à la présence de S, a lieu avant la cause qui est l'émission de S. Le programme n'est donc pas correct du point de vue de la sémantique constructive du langage, que l'on peut étendre au cas des domaines réactifs. Un tel programme sera aussi rejeté par l'analyse de causalité du raffinement d'horloges de QUARTZ [GBS11]. En effet, elle n'autorise le corps d'un domaine qu'à lire ou bien émettre sur un signal lent donné. Il ne peut pas faire les deux à la fois.

Comme dans le cas d'ESTEREL, la sémantique constructive nous permet de ne garder que les programmes « raisonnables ». Elle se base toujours sur les prédicats Must et Can et sur l'alternance entre phases d'exécution et phases de prédiction. Dans l'exemple, S apparait dans la liste Can des signaux potentiellement émis dans l'instant courant de l'horloge globale. On ne peut donc pas décider qu'il est absent. Mais on ne peut pas non plus décider de sa présence puisque le prédicat Must « s'arrête » au niveau du **pause** ck. Autrement dit, on ne peut pas décider de la présence d'un signal tant que l'on n'a pas effectivement exécuté une émission sur ce signal. Le calcul est donc bloqué et le programme n'admet pas de sémantique constructive.

Nous allons utiliser un autre exemple un peu plus complexe pour illustrer à nouveau la sémantique constructive avec les domaines réactifs. Ce programme présente également un problème de causalité :

```
signal S1, S2 in
  domain ck1 do
    present S1 then emit O else nothing end
    ||
    pause ck1; emit S2
  done
  ||
  domain ck2 do
    present S2 then emit O else nothing end
    ||
    pause ck2; emit S1
  done
end
```

On peut remarquer que ce programme est logiquement correct, puisque l'on peut supposer S1, S2 et O présents et ne pas obtenir de contradiction. On va montrer qu'il n'est pas constructivement causal. On commence par exécuter les deux domaines réactifs. Ils sont tous les deux bloqués, puisque le statut des signaux S1 et S2 est encore inconnu. On commence donc une phase de prédiction qui nous permet de déduire que S1, S2 et O peuvent être potentiellement émis. On se retrouve donc bloqué sans pouvoir progresser puisqu'on ne peut décider de l'absence d'aucun signal.

Il est intéressant également de considérer un exemple avec dépendance instantanée accepté par la sémantique constructive :

```
signal S1, S2 in
  domain ck do
    present S1 then pause ck; emit O else nothing end
    ||
    pause ck; emit S2
  done
  ||
  present S2 then emit O end
end
```

La première phase d'exécution ne permet pas d'obtenir d'information sur la présence des signaux. On ne peut pas non plus terminer le premier instant de l'horloge ck. On commence donc une phase de prédiction, qui nous permet de déduire que S1 ne peut pas être émis et est donc absent. Les autres signaux sont potentiellement émis et leur statut reste donc inconnu. On peut donc maintenant effectuer une nouvelle phase d'exécution, qui nous permet d'emprunter la branche **else** du test de présence de S1. On peut donc maintenant terminer le premier instant de ck et exécuter son second instant. On en déduit la présence de S2, puis celle de O.

Implémentation

Nous avons étendu notre petit interprète d'ESTEREL (sans exceptions) pour qu'il prenne en charge les domaines réactifs. Ironiquement, celui-ci n'utilise presque pas de domaines réactifs. On peut justifier ce choix par le fait que l'on ne peut pas faire correspondre un instant d'un domaine réactif en ESTEREL à un instant d'un domaine dans l'interprète puisque l'on souhaite permettre les dépendances instantanées sur un signal lent, ce qui n'est pas possible en REACTIVEML. On utilise toutefois un domaine pour masquer les instants de calcul de l'interprète et faire en sorte que chaque instant de l'horloge globale du programme ESTEREL corresponde à un instant de l'horloge globale de l'interprète. On s'appuie donc sur la version sans domaines que l'on a décrite à la fin de la partie précédente.

Les changements à opérer pour cette version sont plutôt minimes. Comme dans le cas de l'implémentation séquentielle des domaines en REACTIVEML, il suffit de rendre l'implémentation un peu plus modulaire pour permettre l'imbrication des domaines réactifs. On créé donc une structure

de données représentant un domaine réactif et on passe le domaine réactif local en argument des combinateurs. On voit donc que le concept de domaine réactif est naturel, puisque même notre interprète ESTEREL peut le prendre en compte sans problème particulier.

La principale différence de la version avec domaines réactifs est qu'il y a maintenant un signal de prédiction par horloge. Ce signal est reçu par toutes les expressions qui peuvent potentiellement s'exécuter dans l'instant courant de cette horloge. On remplace donc les signaux de prédiction `predict_in` et `predict_out` par des listes associant un signal à une horloge. On modifie ensuite l'opérateur **pause**, qui prend maintenant en argument une horloge `pck`, pour qu'il ne transmette les signaux de prédiction que sur des horloges strictement plus lentes que `pck` :

@ exemples/DoEsteRML/combinators_nodo.rml

```
let process ppause pck lck
      (step_in, predict_in) (step_out, predict_out) =
  ...
  ||
  (* prediction *)
  run par_iter2
    (proc (ck, p_in) (_, p_out) ->
           if ck.c_depth < pck.c_depth then run forward p_in p_out)
    predict_in predict_out
```

Le champ `c_depth` correspond à la profondeur de l'horloge dans l'arbre d'horloges. Les horloges plus lentes que `pck` sont donc celles qui ont une profondeur plus faible. L'autre point important est que pour l'émission d'un signal, on émet le signal `r_may_emit` que lorsque l'on reçoit le signal de prédiction associé à l'horloge du signal. Cela exprime bien le fait que le signal est potentiellement émis pendant l'instant courant de son horloge. De même, dans le cas du **present**, il ne faut prendre en compte le statut du signal que si l'on est sûr qu'il s'agit bien du même instant de l'horloge du signal.

CHAPITRE 11

Conclusion et Perspectives

Dans ce chapitre, nous allons présenter plusieurs extensions et pistes d'améliorations à court ou moyen terme. Celles-ci concernent le langage, les analyses statiques et systèmes de types ainsi que les différentes implémentations. Ces extensions n'ont pour la plupart pas encore été implémentées et constituent des pistes de réflexion. Nous conclurons enfin par un résumé du manuscrit et des perspectives à plus long terme et des axes de recherche.

11.1 Extensions et perspectives

Langage

Extensions des domaines réactifs Nous nous sommes restreints dans ce manuscrit aux opérateurs et constructions les plus importantes et les plus fondamentales du langage. Cela permet de simplifier la présentation du langage, notamment sa sémantique. Nous avons également cherché à ne pas ajouter trop de fonctionnalités superflues tant que leur utilité n'a pas été prouvée. Un bon exemple de ce choix est la construction `by` pour créer un domaine périodique. Elle accepte une expression quelconque, mais qui est évaluée une bonne fois pour toute au moment de la création du domaine. Le nombre d'instants du domaine est alors constant pendant toute sa durée de vie. Il serait possible d'ajouter une variante où l'expression donnant le nombre d'instants est évaluée à chaque instant. On pourrait par exemple la noter avec le mot-clé `by?` :

```
let process domain_not_constant p =
  domain ck by? last p + 1 do ... done
```

On utilise ici la valeur précédente du signal `p` pour calculer le nombre d'instants du domaine réactif, qui peut donc varier à chaque instant de son horloge parente. Tous les exemples que nous avons écrits utilisent soit un domaine non borné, soit une borne constante. Il reste donc à trouver un exemple pour motiver l'ajout de cette construction.

Une autre modification plus intéressante concerne les domaines réactifs périodiques. Lorsque le corps d'un domaine de période `n` boucle sur l'horloge locale, le domaine attend automatiquement l'instant suivant de son horloge parente après avoir exécuté `n` instants locaux. Il n'attend jamais une horloge plus lente. Le problème est que cela rend le domaine réactif parent non coopératif si celui-ci n'est pas aussi périodique. Par exemple, le processus suivant n'est pas réactif, puisque le domaine d'horloge `ck2` n'est pas coopératif :

```
let process nested_domains =
  domain ck2 do
    domain ck1 by 1 do
      loop pause ck1 end
    done
  done
```

Le domaine réactif d'horloge `ck1` est équivalent à **`loop pause`** `ck2` **`end`**, ce qui rend son domaine parent non coopératif.

Une première solution à ce problème serait d'introduire une variante **`quiet`** des domaines réactifs. Comme dans le cas de **`quiet pause`**, un tel domaine serait considéré en attente par son domaine parent. Une solution alternative est de choisir l'horloge qu'attend le domaine une fois sa période écoulée, pour pouvoir attendre une horloge plus lente. On pourrait écrire :

```
domain ck by n each lck do ... done
```

La sémantique de cette construction est que le domaine réactif exécute au maximum `n` instants par instant de l'horloge `lck`. La construction actuelle correspond donc au cas où l'on prend `lck = local_ck`. L'autre avantage de cette version généralisée est que son comportement est indépendant de l'horloge locale, de la même manière que **`pause`** `ck` attend le prochain instant de l'horloge `ck` quelle que soit l'horloge locale. Pour donner la sémantique formelle de cette construction, on ajoute cette horloge dans l'expression représentant le domaine en cours d'exécution. On note maintenant e_1 `in` $ck'[i/j/lck]$, où ck' est l'horloge du domaine, e_1 le corps du domaine, i représente le nombre d'instants effectués dans l'instant courant de l'horloge parente, j le nombre maximal d'instants par instant de l'horloge parente et lck est l'horloge que doit attendre le domaine lorsque $i = j - 1$. Pour la sémantique opérationnelle, on doit modifier la définition du prédicat $\vdash_{\text{next}}$ (figure 5.1 page 76) et de la règle PARENTEOI (figure 5.3b page 79) :

$$\text{(NEXTIN')}\ \frac{i < j - 1 \qquad s :: (ck', i), C \cup \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1}{s, C, \mathcal{S} \vdash_{\text{next}} e_1\ \texttt{in}\ ck'[i/j/lck]}$$

$$\text{(NEXTINCOUNTER)}\ \frac{s :: (ck', i), C \cup \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1 \qquad lck \in C}{s, C, \mathcal{S} \vdash_{\text{next}} e_1\ \texttt{in}\ ck'[j-1/j/lck]}$$

$$\text{(PARENTEOI')}\ \frac{\begin{array}{c} s' = s :: (ck', i) \qquad C' = C \cup \{ck'\} \\ not(i < j - 1 \wedge s', \{ck'\}, \mathcal{S} \vdash_{\text{next}} e_1) \qquad s', C' \vdash e_1/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'/\mathcal{S}' \\ \mathcal{S}'' = \mathcal{S}' \sqcup \{\texttt{snext}(s', C') \mapsto \texttt{next}_{C'}(\mathcal{S}(s'))\} \qquad i' = \texttt{ if } lck \in C \texttt{ then } 0 \texttt{ else } i + 1 \end{array}}{s, C \vdash e_1\ \texttt{in}\ ck'[i/j/lck]/\mathcal{S} \longrightarrow_{\text{eoi}} e_1'\ \texttt{in}\ ck'[i'/j/lck]/\mathcal{S}''}$$

Dans le cas du prédicat $\vdash_{\text{next}}$, on procède de la même façon qu'avant tant que le compteur n'a pas atteint sa valeur maximale (NEXTIN'). Lorsque le compteur atteint $j - 1$, le domaine attend le prochain instant de lck (NEXTINCOUNTER). Cela se traduit également dans la règle PARENTEOI' par le fait que l'on ne remet à zéro le compteur du domaine que lorsque l'on est dans la fin de l'instant de l'horloge lck. On peut remarquer que dans le cas où lck est l'horloge locale, la condition $lck \in C$ est toujours vérifiée. On retombe donc bien sur la sémantique que l'on a vue dans le chapitre 5. On peut également adapter de façon similaire la sémantique comportementale, ainsi que l'analyse de réactivité (on peut trouver la règle originale dans la figure 7.1 page 119) :

$$\text{(INPERIOD')}\ \frac{\Gamma, ck' \vdash e_1 : rt \mid \kappa_1 \qquad [] \vdash \kappa_1}{\Gamma, ce \vdash e_1\ \texttt{in}\ ck'[i/j/ce''] : rt \mid \texttt{domain}\ (\kappa_1[\gamma \leftarrow 0 \mid ce''])}$$

On remplace l'horloge locale γ par l'horloge ce'' que doit attendre le domaine. Là encore, on retombe bien sur la règle originale si l'on prend `lck = local_ck`.

Opérateurs Il peut être utile pour simplifier l'écriture des programmes de pouvoir accéder à l'horloge d'un signal. On définit pour cela dans la librairie standard l'opérateur `clock_of` :

```
val clock_of : ('a, 'b) event{'ck|} -> ['ck]
```

L'implémentation de cette primitive ne pose aucun problème puisque l'on doit de toute façon stocker l'horloge du signal pour savoir par exemple quelle fin d'instant attendre. On peut utiliser cette fonction pour simplifier l'écriture du processus `quiet_sustain_v` (page 39) qui maintient la

valeur d'un signal. Celui-ci attend l'instant suivant de l'horloge du signal pour émettre sa valeur précédente :

```
let process quiet_sustain_v s =
  loop
    emit s (last s);
    quiet pause (clock_of s)
  end
```

On pourrait également vouloir ajouter des opérateurs pour manipuler les horloges, comme par exemple l'opérateur `parent_clock` qui renvoie l'horloge parente d'une horloge :

```
val parent_clock : ['ck] -> ['ck2]
```

Le type de cette primitive pose problème, puisqu'il n'y a aucun lien entre les deux horloges. Notre calcul d'horloges ne garde en effet aucune relation entre les horloges. Il ne peut donc pas garantir que l'horloge de sortie est bien l'horloge parente de l'horloge d'entrée. Le type de l'horloge de sortie est donc non contraint, un peu à la manière de la fonction `magic` du module `Obj`[1] d'OCaml. On choisit donc de ne pas ajouter cette primitive dans le langage. Puisque les horloges sont des objets de première classe, on peut se passer de cette fonction et transmettre l'horloge parente par exemple comme argument d'un processus.

Domaines réactifs asynchrones Une autre extension est d'utiliser les domaines réactifs avec d'autres relations entre les horloges. Nous avons montré une approche où les instants d'un domaine réactif sont inclus dans les instants de l'horloge parente. A l'opposé, on peut imaginer que les instants d'un domaine réactif soient complètement désynchronisés avec ceux de l'horloge parente. On se rapproche alors de la notion de *futur* [HJ85], qui est l'un des moyens classiques de paralléliser un programme. Un futur représente une promesse de résultat, que l'on ne lit que lorsqu'on a réellement besoin de la valeur. On peut donc lancer le calcul de cette valeur dans un thread séparé. Un exemple typique d'utilisation des futurs consiste à lancer une tâche longue mais rare dans un thread séparé pour ne pas ralentir le reste du programme. C'est particulièrement utile dans le cas des systèmes réactifs ou interactifs. Par exemple, on peut continuer à mettre à jour une interface graphique pendant qu'un calcul long se déroule en parallèle. [CGP12] montre l'intérêt des futurs pour la programmation des systèmes embarqués dans un langage synchrone flot de données.

Une première idée consiste à pouvoir lancer une fonction dans un thread séparé. On obtient alors un futur représentant le résultat de la fonction, que l'on peut obtenir lorsque l'on en a vraiment besoin. L'opérateur `exec` d'Esterel [BRS93] permet d'obtenir un résultat similaire, en attendant la fin du calcul. Mais on peut aussi remplacer cette fonction par un domaine réactif avec sa propre horloge. Il peut donc être constitué de multiples processus communiquant et se synchronisant en utilisant toutes les fonctionnalités du langage. On pourrait écrire :

```
let process domain_async =
  let f = domain ck async do run do_stuff done in
  pause;
  pause;
  print_int (get f)
```

On lance le processus `do_stuff` qui effectue un calcul long dans un domaine réactif asynchrone. On obtient alors un futur `f`, puis on continue l'exécution du programme. Après plusieurs instants, on appelle l'opérateur `get` pour obtenir la valeur calculée par `do_stuff`. Cette opération bloque tant que le résultat n'est pas disponible. Si le processus `do_stuff` a le type `'a process`, alors `f` a le type `'a futur` et l'opérateur `get` a le type `'a futur -> 'a`. On peut également attendre le résultat du calcul comme on attend l'émission d'un signal avec une construction que l'on pourrait appeler `await futur`.

La principale question que pose cette construction est de savoir si l'on permet de communiquer entre un domaine réactif asynchrone et le reste du programme et si oui dans quelles conditions. Le

1. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Obj.html

choix le plus conservateur est d'interdire complètement toute communication avec l'extérieur. Il suffit alors de forcer l'effet du domaine asynchrone à être nul, c'est-à-dire que le corps du domaine ne peut pas accéder à des signaux définis en dehors du domaine :

$$\frac{\Gamma[x \mapsto \{\gamma\}], \gamma \vdash e_1 : ct \mid \{\gamma\} \qquad \gamma \notin \mathit{ftv}(\Gamma, ct)}{\Gamma, ce \vdash \texttt{domain}\ x\ \texttt{async}\ e_1 : ct \mid \emptyset}$$

Un autre choix consiste à permettre les communications asynchrones entre domaines. On pourrait s'inspirer de ce qui est fait dans les FAIRTHREADS [Bou06] et FUNLOFT [BD08]. On perd alors les propriétés de déterminisme et de reproductibilité du langage.

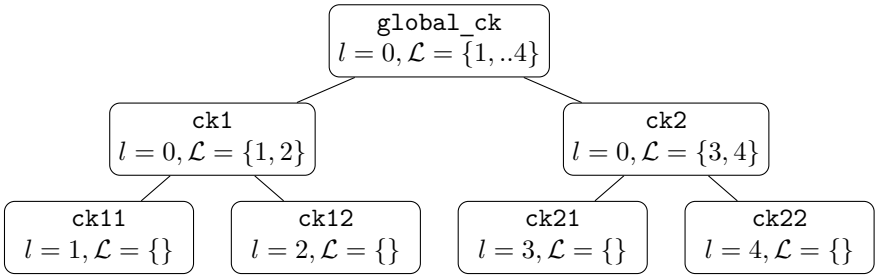
L'implémentation de cette construction ne devrait pas poser de problèmes. Il faut prendre soin à ce que les futurs soient compatibles avec l'ordonnancement coopératif. Cela signifie que lorsque l'on cherche à obtenir la valeur d'un futur avec l'opérateur `get`, il ne faut pas que le processus bloque réellement en attendant la fin du calcul du domaine asynchrone. Comme pour les signaux, on doit mettre le processus dans une liste d'attente et le réveiller une fois la valeur disponible.

Implémentation

Équilibrage de charge L'exécution parallèle et distribuée de programmes REACTIVEML pose la question de la répartition du calcul entre les différents processus. Dans le cas du moteur d'exécution parallèle en mémoire partagée, on utilise le vol de tâches pour répartir le travail de façon dynamique entre les threads. Dans la version OCAML avec envoi de messages, les domaines réactifs sont répartis au moment de leur création entre les différents processus et ne peuvent pas migrer par la suite. Il faut donc choisir au moment où l'on crée un domaine réactif si l'on souhaite l'exécuter localement ou bien sur un processus distant, et si oui lequel.

La répartition des domaines réactifs doit suivre plusieurs règles. En effet, on a vu que plusieurs opérations, notamment l'obtention d'un signal par l'envoi du message `Mreq_signal`, sont bloquantes. Cela signifie que le processus qui fait la requête reste bloqué en attendant la réponse du propriétaire du signal. Il faut éviter la situation de *deadlock*, où plusieurs threads s'attendent mutuellement et restent bloqués pour toujours. Pour cela, on décide qu'un processus ne peut accueillir que les domaines réactifs avec le même domaine parent.

On associe donc à chaque domaine réactif un ensemble $\mathcal{L}$ de processus sur lesquels peuvent s'exécuter ses domaines enfants. Cet ensemble est subdivisé en un ensemble de processus sur lesquels s'exécutent les enfants directs du domaine et un ensemble de processus sur lesquels s'exécutent les descendants des enfants. Ce second ensemble est l'union disjointe des ensembles $\mathcal{L}_i$ associé aux domaines réactifs enfants. Supposons par exemple que l'on dispose de 5 processus. On peut alors répartir l'arbre d'horloges suivant selon les valeurs indiquées sur les nœuds ($l = i$ indique que le domaine réactif s'exécute dans le processus i) :



Plusieurs variantes d'équilibreurs de charge ont été écrits. On peut choisir celui qui est utilisé avec l'option `-balance` du programme généré. Un premier équilibreur appelé `local` lance tous les domaines réactifs dans le processus maitre. A l'opposé, `remote` exécute tous les domaines réactifs sur un processus distant, ce qui est très rarement un bon choix. Au contraire, cela a beaucoup aidé à la mise au point du moteur d'exécution en se plaçant dans un cas où les communications entre processus sont nombreuses. L'équilibreur de charge standard, noté `robin`, lance le premier domaine réactif localement puis les suivants sur des processus distants, suivant un *round robin*.

Enfin, l'utilisateur peut influencer le choix de l'équilibreur à l'aide d'une annotation donnée avec le mot-clé **schedule** lors de la définition du domaine. Il s'agit d'une fonction prenant en entrée le nombre de processus alloués au domaine parent encore disponibles et qui renvoie deux entiers. Le premier donne le numéro du processus où doit s'exécuter le domaine, sachant que 0 est le processus local. Le second est le nombre de processus à allouer aux enfants de ce domaine (dans la limite des processus disponibles). On obtient le résultat visible sur l'arbre ci-dessus en écrivant :

```
let process domain_schedule =
  domain ck1 schedule (fun nb -> 0, nb/2) do
    domain ck11 schedule (fun nb -> 1, 0)
    ||
    domain ck12 schedule (fun nb -> 2, 0)
  done
  ||
  domain ck2 schedule (fun nb -> 0, nb/2)  do
    domain ck21 schedule (fun nb -> 1, 0)
    ||
    domain ck22 schedule (fun nb -> 2, 0)
  done
```

Région d'un signal Nous avons vu dans le cas du moteur d'exécution parallèle que les signaux étaient gérés par le processus dans lequel s'exécute le domaine réactif auquel est attaché le signal. Mais ceci n'est pas forcément utile si jamais le signal n'est utilisé que localement. On peut dans ce cas gérer le signal localement et éviter ainsi de nombreux envois de messages. Cette information qu'un signal n'est jamais utilisé en dehors d'un domaine réactif (dont l'horloge est plus rapide que celle du signal) peut également être utile dans le cas du moteur parallèle en mémoire partagée, puisqu'elle peut permettre de se passer de verrous pour ce signal.

On ajoute donc un nouvel argument au moment de la définition d'un signal. On donne après le mot-clé **region** une horloge qui indique que le signal n'est pas utilisé en dehors du domaine associé à cette horloge, que l'on appelle sa région. La région d'un signal est égale à son horloge si jamais on omet cette annotation. Si la région est plus lente que l'horloge du signal, alors l'annotation est inutile puisque le signal ne peut de toute façon pas échapper du domaine attaché à son horloge. On peut par exemple écrire :

```
let process signal_restrict =
  domain ck do
    signal s clock global_ck region ck in
    ...
  done
```

Dans le moteur d'exécution distribué, le signal `s` sera donc géré par le processus où s'exécute le domaine réactif d'horloge `ck`. Il n'est pas nécessaire de communiquer sa valeur aux autres processus. Cette fonctionnalité est implémentée dans la version actuelle du moteur d'exécution.

Le calcul d'horloges peut être étendu pour garantir qu'un signal n'échappe jamais de sa région. On ajoute pour cela la région ce_r dans le type (α_1, α_2) `event`$\{ce, ce_r|\}$ d'un signal. La notation précédente est simplement un raccourci pour :

$$(\alpha_1, \alpha_2)\ \texttt{event}\{ce|\} \triangleq (\alpha_1, \alpha_2)\ \texttt{event}\{ce, ce|\}$$

On modifie ensuite le calcul d'horloges pour prendre en compte la région, qui doit apparaître dans l'effet associé à chaque accès au signal :

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_{ck} : \{ce'\} \mid \emptyset \qquad \Gamma, ce \vdash e_r : \{ce_r\} \mid \emptyset \\ \Gamma, ce \vdash e_d : ct_2 \mid \emptyset \qquad \Gamma, ce \vdash e_g : ct_1 \xrightarrow{\emptyset} ct_2 \xrightarrow{\emptyset} ct_2 \mid \emptyset \\ \Gamma, ce \vdash e_{rck} : \{ce''\} \mid \emptyset \qquad \Gamma[x \mapsto (ct_1, ct_2)\ \texttt{event}\{ce', ce_r|\}], ce \vdash e_1 : ct \mid cf_1\end{array}}{\Gamma, ce \vdash \texttt{signal}\ x\ (\texttt{h} : b, \texttt{ck} : e_{ck}, \texttt{r} : e_r, \texttt{d} : e_d, \texttt{g} : e_g, \texttt{rck} : e_{rck})\ \texttt{in}\ e_1 : ct \mid cf_1 \cup \{ce'; ce''; ce_r\}}$$

$$\frac{\Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e_s : (ct_1, ct_2)\ \texttt{event}\{ce', ce_r|\} \mid \emptyset \qquad \Gamma[x \mapsto ct_2], ce \vdash e_2 : ct \mid cf_2}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{until}\ e_s(x) \to e_2 : ct \mid cf_1 \cup cf_2 \cup \{ce'\} \cup \{ce_r\}}$$

On modifie de façon similaire les autres règles ainsi que la version avec rangées du calcul d'horloges. On peut remarquer que l'on ne garantit pas que la région est plus rapide que l'horloge du signal, mais seulement que le signal n'échappe ni de son horloge ni de sa région. Le système de types ne peut pas en effet parler des relations entre les horloges. Cela n'est pas très grave puisque si jamais la région d'un signal est strictement plus lente que son horloge, alors cette annotation est simplement inutile.

Échappement de portée des signaux Nous avons déjà décrit deux systèmes de types-et-effets appliqués à REACTIVEML, pour le calcul d'horloges et pour l'analyse de réactivité. Cette famille de systèmes de types est un outil important pour les langages fonctionnels avec effets de bords comme REACTIVEML, qui a sans doute d'autres applications pour ce langage. On peut par exemple appliquer *l'analyse d'effets de bords* de [NN99] pour détecter l'échappement de portée des signaux. Il s'agit de détecter si un signal échappe de sa portée lexicale, ce qui permet de remplacer un signal par une simple référence (si on peut ordonnancer le programme statiquement). Une première approche basée sur l'interprétation abstraite a déjà été expérimentée sans succès dans [Gay08]. L'utilisation d'un système de types-et-effets permet d'obtenir une analyse plus simple à implémenter, mais qui est possiblement moins précise.

L'idée de l'analyse est associer à chaque définition de signal une étiquette notée π. Le type d'un signal contient alors son étiquette (ou bien une rangée d'étiquettes). L'effet d'une expression est l'ensemble des étiquettes des signaux auxquels elle accède. On teste l'échappement de portée en vérifiant si l'étiquette du signal apparait libre dans le type de retour du corps de la définition du signal ou dans l'environnement de typage, tout comme on l'a fait pour les domaines réactifs dans le calcul d'horloges :

$$\frac{\Gamma, ce \vdash e_1 \rightsquigarrow e_1' : (ct_1, ct_2)\ \texttt{event}_\pi\{ce'|\} \mid \emptyset \qquad \Gamma, ce \vdash e_2 \rightsquigarrow e_2' : ct_1 \mid \emptyset}{\Gamma, ce \vdash \texttt{emit}\ e_1\ e_2 \rightsquigarrow \texttt{emit}\ e_1'\ e_2' : \texttt{unit} \mid \{\pi\}}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 \rightsquigarrow e_1' : ct \mid cf \\ e' = \texttt{if}\ \pi \notin ftv(\Gamma, ct)\ \texttt{then}\ (\texttt{static signal}\ {}_\pi s\ \texttt{in}\ e_1')\ \texttt{else}\ (\texttt{signal}\ {}_\pi s\ \texttt{in}\ e_1')\end{array}}{\Gamma, ce \vdash \texttt{signal}\ {}_\pi s\ \texttt{in}\ e_1 \rightsquigarrow e' : ct \mid cf}$$

On peut voir dans les règles que l'on réécrit le programme pendant le typage. En particulier, on transforme l'expression $\texttt{signal}\ {}_\pi s\ \texttt{in}\ e_1$ en $\texttt{static signal}\ {}_\pi s\ \texttt{in}\ e_1'$ si le nom π n'apparait ni dans le type de retour ct, ni dans l'environnement Γ. On se rappelle ainsi du fait que le signal n'échappe pas de sa portée.

Debugging On a vu que l'ajout des domaines réactifs dans le langage pouvait rendre un programme non coopératif. Nous avons présenté dans le chapitre 7 une analyse statique permettant de détecter les domaines réactifs potentiellement non coopératifs. Malgré tout, cette analyse a des limites et signale parfois des faux positifs, en particulier lorsque la réactivité du programme dépend de valeurs. Il arrive donc que l'on lance un programme que l'on pense être réactif, pour lequel on ignore donc les avertissements de l'analyse de réactivité, mais qui boucle indéfiniment. Il est alors

complexe de comprendre comment se déroule l'exécution du programme, en particulier lorsque le programme compte plusieurs niveaux d'horloges. L'interprète ESTEREL que nous avons décrit dans le chapitre 10 en est un bon exemple.

La réponse à ce problème est bien entendu un debugger de programmes REACTIVEML. Nous nous intéressons ici uniquement au cas des domaines réactifs. Un bon outil pour détecter les domaines non coopératifs est la possibilité de connaître les instants qu'exécute chaque domaine et de pouvoir choisir quel domaine exécuter. On peut ainsi imaginer un moteur d'exécution dans lequel plutôt que d'exécuter un domaine réactif, on le place dans une file d'attente globale. L'utilisateur peut alors choisir quel domaine il souhaite exécuter, en choisissant par exemple le nombre d'instants. Il pourrait également être intéressant de savoir quels processus souhaitent exécuter un autre instant de l'horloge locale. Cela suppose de garder le lien entre les processus exécutés et le code source.

11.2 Conclusion

Résumé

Nous avons présenté dans ce manuscrit deux contributions principales : une extension du modèle de concurrence synchrone réactif pour le raffinement temporel et une implémentation parallèle de REACTIVEML. Ces deux contributions ne sont pas sans liens, puisque le concept de domaine réactif favorise une exécution parallèle efficace, en plus des gains d'expressivité et de modularité qu'il procure. Nous avons également développé les outils théoriques pour pouvoir définir formellement la sémantique des nouvelles constructions, ainsi que les analyses statiques permettant de garantir la sûreté des programmes.

Domaines réactifs La première contribution de ce manuscrit, qui est aussi la plus importante selon nous, est la définition des domaines réactifs. Un domaine réactif crée une notion d'instant local invisible de l'extérieur. Nous avons tenté, au cours de ce manuscrit, de montrer l'utilité pratique des domaines réactifs au travers de nombreux exemples. Nous avons notamment mis en pratique ces nouveautés pour la programmation d'un interprète ESTEREL.

L'avantage des domaines réactifs est la relative simplicité de leur implémentation, puisqu'il s'agit d'une réification du moteur d'exécution. Cela signifie qu'un domaine réactif correspond à un programme REACTIVEML lancé à l'intérieur d'un autre programme REACTIVEML, avec son propre rythme local et qui peut communiquer avec son hôte. Plus pragmatiquement, les fonctionnalités des domaines réactifs comme l'attente automatique du domaine parent sont fortement inspirées par le moteur d'exécution, ce qui explique que leur implémentation soit simple. Nous avons aussi montré comment on pouvait étendre les sémantiques comportementales et opérationnelles à ce cadre étendu, en conservant les principes des versions précédentes.

Bien qu'il s'agisse d'une avancée importante pour REACTIVEML, les domaines réactifs apportent une certaine complexité au langage. Comprendre un programme avec plusieurs niveaux d'horloges et l'attente automatique des domaines peut devenir complexe. L'absence d'utilisateurs extérieurs laisse en suspens la question de l'utilisation de cette extension de REACTIVEML par des programmeurs non experts. Il s'agit, sans doute, d'une des déceptions de ce travail orienté vers la programmation. En outre, le nombre de constructions ajoutées au langage peut sembler un peu trop élevé, notamment en ce qui concerne les variantes de constructions comme `quiet pause`. Celles-ci nous semblent nécessaires, faute de solution plus élégante.

Les domaines réactifs rendent le modèle synchrone plus modulaire, en permettant de cacher des instants liés à des communications locales. Le raffinement temporel, c'est-à-dire la possibilité de remplacer une approximation instantanée d'un système par une version plus complexe que l'on simule en plusieurs instants, devient également aisé. La possibilité de masquer des instants nous apparait comme une extension naturelle du modèle synchrone, complémentaire du sous-échantillonnage. Il serait donc intéressant d'étudier l'ajout de la notion de domaines réactifs dans d'autres langages synchrones. Nous avons déjà suggéré la possibilité d'intégrer les domaines réactifs dans ESTEREL en s'inspirant des travaux sur la sémantique constructive. On pourrait également se poser la même question dans le cas de langages synchrones flot de données.

Systèmes de types La seconde contribution de ce manuscrit est une conséquence de la première. Il s'agit de deux systèmes de types-et-effets permettant de garantir la sûreté des programmes. Le premier est un calcul d'horloges qui empêche une utilisation incorrecte des signaux et des horloges, puisqu'un signal ne peut pas être utilisé en dehors du domaine auquel il est attaché. Le second est une analyse de réactivité qui détecte les boucles potentiellement instantanées. Son approche est suffisamment générique pour s'appliquer aussi bien à REACTIVEML classique, qu'au cas des domaines réactifs et probablement à d'autres modèles de concurrence. Nous avons ensuite étendu ces deux systèmes de types avec une approche nouvelle du *subeffecting* inspirée des types rangées. Nous avons aussi montré la sûreté de ces deux systèmes de types vis-à-vis de la sémantique formelle.

L'absence de polymorphisme de rang supérieur rend nécessaire la présence d'annotations de typage et empêche parfois l'écriture de certains programmes. On a vu dans le cas de l'interprète ESTEREL du chapitre 10 que l'impossibilité de décrire un processus ayant un effet sur au moins une horloge donnée posait aussi des problèmes.

Le calcul d'horloges avec rangées remplit bien son rôle puisqu'il permet d'accepter la grande majorité des programmes que l'on souhaite écrire. L'analyse de réactivité présente elle un bon compromis entre la simplicité et la précision de l'analyse. Les deux systèmes de types utilisent des techniques classiques et peuvent ainsi s'intégrer simplement dans le compilateur. Un axe de recherche est la définition d'un système de types avec polymorphisme de rang supérieur pour les horloges et permettant une meilleur expressivité dans les annotations de typage.

Implémentation parallèle et distribuée La dernière contribution est l'implémentation parallèle du langage. Nous avons présenté deux approches différentes. La première utilise des threads communiquant par mémoire partagée, ce qui permet des modifications mineures sur le moteur d'exécution. Elle montre des possibilités de gains intéressants, bien qu'il ne s'agisse que d'une première version qui peut encore être optimisée. Cependant, le gain par rapport à l'implémentation de référence en OCAML est finalement faible. En outre, l'obligation d'utiliser le langage F# pour utiliser cette version qui ne fonctionne que sous Windows rend son intérêt pratique limité.

La seconde implémentation du moteur d'exécution utilise des processus lourds communiquant par envoi de messages avec la bibliothèque MPI. Les résultats expérimentaux sont dans ce cas décevants, puisqu'il faut une grande quantité de calcul par rapport aux communications pour voir un gain.

Ces expériences montrent que l'exécution parallèle de programmes REACTIVEML peut offrir des gains intéressants. La solution à terme est sans doute de combiner les deux approches, en utilisant les threads localement et l'envoi de messages pour la communication sur un réseau. L'interaction entre l'exécution parallèle et le garbage collector est un des problèmes à résoudre.

Perspectives à long terme

Nous concluons en donnant un aperçu de ce que pourrait être le futur de REACTIVEML. L'idée maitresse est de voir REACTIVEML comme une extension d'OCAML, comme l'est par exemple JOCAML, au lieu d'un compilateur séparé. Cela augmenterait l'expressivité du langage en permettant d'utiliser toutes les fonctionnalités d'OCAML comme les foncteurs ou les objets. Le compilateur actuel serait toujours utile comme terrain de jeu pour expérimenter des extensions du langage.

On peut aussi considérer la nécessité d'intégrer dans OCAML et son moteur d'exécution les briques de base pour une extension concurrente du langage. Le point le plus important est bien entendu un garbage collector parallèle efficace. En outre, un ordonnanceur avec vol de tâches doit sans doute être intégré directement au moteur d'exécution tant ses liens avec le garbage collector et le fonctionnement interne du moteur sont importants. Il pourrait ensuite être utilisé par les différentes extensions du langage, qui proposent chacune leur modèle de concurrence et dont fait partie REACTIVEML. L'utilisateur est alors libre de choisir le modèle le plus adapté à son problème. Cela pose également des questions sur l'organisation du compilateur ou du système de types pour rendre l'écriture de ces extensions aisée, sans avoir besoin de modifier le cœur du compilateur.

Annexes

ANNEXE **A**

Encodage des constructions

Dans ce court chapitre, nous allons montrer des encodages des différentes constructions que nous avons présentées dans ce manuscrit, c'est-à-dire les signaux à mémoire et les domaines réactifs. Il s'agit d'encodage parmi d'autres, qui permettent de montrer la possibilité de cet encodage, mais ils ne sont pas nécessairement minimaux. C'est en effet une question légitime que l'on peut se poser à chaque fois que l'on souhaite ajouter une nouvelle construction, en particulier dans un langage aussi expressif que REACTIVEML. Nous allons voir que dans les deux cas, un tel encodage est possible. Cela montre bien que ces constructions sont des extensions naturelles du langage et permet d'en avoir une vision plus opérationnelle. La taille et la complexité de ces encodages sont toutefois des argument forts pour leur intégration dans le langage. L'autre argument est bien entendu la correction, puisque ces encodages requièrent de suivre plusieurs règles implicites pour obtenir le résultat attendu, alors que l'on peut les garantir statiquement une fois ces constructions ajoutées dans le langage.

A.1 Un encodage des signaux à mémoires

Les signaux à mémoire en REACTIVEML sans domaines

Dans le cas de l'exemple de la banque (page 22), le maintien du signal est très simple puisque la fonction de combinaison (c'est-à-dire l'addition) est commutative et associative. Il suffit donc juste d'émettre la valeur précédente du signal à chaque instant. Si la fonction de combinaison n'est pas commutative, alors le maintien de la valeur en utilisant un signal normal demande plus de travail. La figure A.1 montre un encodage générique, pour n'importe quel signal de type `('a, 'b) event`. On rappelle que `'a` est le type des valeurs émises, alors que `'b` est celui de la valeur du signal. La valeur par défaut `d` est donc de type `'b`, alors que la fonction de combinaison `g` a le type `'a -> 'b -> 'b`.

On définit tout d'abord le type `('a, 'b) value` des valeurs stockées dans le signal. L'idée est de distinguer la valeur par défaut du signal des autres valeurs, afin de savoir s'il s'agit de la première émission sur le signal, auquel cas il faut utiliser la valeur précédente du signal pour la combinaison. La valeur stockée est donc soit `Empty d` (où `d` est la valeur par défaut du signal) pour indiquer qu'aucune valeur n'a encore été émise, soit `Acc acc` où `acc` est la valeur courante accumulée dans le signal. La valeur réelle du signal à mémoire s'obtient avec la fonction `last_` (ligne 3). Si aucune valeur n'a encore été émise sur le signal, on obtient alors `Empty d` et on renvoie donc la valeur par défaut du signal. Sinon, on renvoie simplement la valeur dans l'accumulateur. L'émission consiste à émettre un couple formé de la valeur et de l'ancienne valeur du signal. S'il s'agit de la première valeur émise dans l'instant, c'est-à-dire si le premier argument `old` de la fonction de combinaison est égal à `Empty _`, alors il faut en fait donner la valeur précédente à la fonction de combinaison `g` du signal à mémoire (lignes 12-13). Sinon, on applique normalement la fonction de combinaison du signal (ligne 14).

exemples/annA/memory_last.rml

```
type ('a, 'b) value = Empty of 'b | Acc of 'b

let last_ s =
  let v = last s in
  match v with
  | Empty d -> d
  | Acc acc -> acc

let emit_ s v = emit s (last s, v)

let gather_ g (l, v) old = match old, l with
| Empty _, Empty d -> Acc (g v d)
| Empty _, Acc l -> Acc (g v l)
| Acc acc, _ -> Acc (g v acc)

let process mk_memory d g =
  signal s default Empty d gather gather_ g in s
```

FIGURE A.1 – Encodage générique des signaux à mémoires

On peut remarquer que cet encodage est plutôt simple et efficace. Mais l'implémentation des signaux à mémoire est encore plus simple et leur utilisation tellement courante que leur ajout dans le langage se justifie totalement.

Réinitialisation des signaux

L'autre extension des signaux que l'on a présentée est la possibilité de réinitialiser la dernière valeur d'un signal à chaque nouvel instant d'une horloge. Nous allons montrer que nous pouvons encoder cette fonctionnalité dans certains cas par une transformation de programmes. Nous allons illustrer cet encodage sur le processus suivant :

```
let process memory_reset =
  domain ck do
    memory m default 0 gather (fun f acc -> f acc) reset global_ck in
    m <<- (+) 4;
    pause global_ck;
    m <<- (+) 2;
    pause ck;
    print_int (last m)
  done
```

Ce programme affiche `2`, puisque la valeur précédente du signal est réinitialisée à `0` à la fin du premier instant de l'horloge globale. La première modification du signal à mémoire est donc oubliée.

Pour encoder cette réinitialisation, on va s'inspirer d'un des exemples typiques de programmation avec les domaines : la possibilité d'ordonner des actions dans un instant (page 42). L'idée est d'ajouter un nouveau domaine réactif d'horloge `rck` afin de subdiviser chaque instant de l'horloge de réinitialisation du signal, ici l'horloge globale. Le programme va s'exécuter dans le premier instant de ce domaine, alors que la réinitialisation de la mémoire a lieu dans le second instant. Le programme modifié devient :

```
let process memory_reset_encode =
  domain rck do
    domain ck do
      memory m default 0 gather (fun f acc -> f acc) in
      loop
        pause rck;
        m <== 0;
        pause global_ck
      end
      ||
      m <<- (+) 4;
      ...
```

On remarque que cet encodage requiert une transformation du programme non locale et donc non modulaire. En outre, il ne fonctionne que si aucun processus n'attend l'émission du signal au moment de sa réinitialisation, car on doit émettre une valeur sur le signal pour changer sa dernière valeur. Cet encodage ne marche pas non plus si le domaine d'horloge `ck` est périodique, puisque l'on change son horloge parente. L'ajout de cette construction dans le langage permet de lever toutes ces limitations.

A.2 Un encodage partiel des domaines réactifs

Nous allons maintenant présenter un encodage partiel des domaines réactifs. Il reprend plusieurs des idées que l'on a esquissé à la fin du chapitre 2. Notre but n'est pas de proposer une traduction fidèle de la sémantique formelle mais de donner une intuition opérationnelle des domaines réactifs. Il existe probablement des cas où cette traduction est incorrecte.

Domaines réactifs

Structure de données On encode une horloge par un type enregistrement contenant les champs suivants :

- Le nom de l'horloge.
- Un signal `step` émis au début de chaque instant de ce domaine.
- Un signal `eoi` émis à la fin de chaque instant de ce domaine.
- Un signal `has_next` qui sera émis pendant la fin de l'instant du domaine si jamais il est nécessaire de faire un autre instant local. Si ce signal est absent, le domaine attend automatiquement le prochain instant de son domaine parent.
- Un signal `alive` qui compte le nombre de domaines réactifs enfants en cours d'exécution.
- Un signal `hold` pour le maintien des signaux. Nous en parlerons plus en détail dans la partie suivante.
- Un signal `finished` qui permet de terminer l'exécution du domaine.

Fonctionnement Le fonctionnement d'un domaine réactif est donné par le processus `_domain` visible sur la figure A.2 :

- Il commence l'exécution d'un instant local en émettant le signal `step` (ligne 9).
- Il attend la fin de l'instant, c'est-à-dire l'exécution des processus dans le domaine et de ses domaines enfants grâce au signal `alive`.
- Il émet ensuite le signal `eoi` (ligne 12) pour signaler la fin de l'instant de cette horloge. Les processus en attente d'un prochain instant de cette horloge, par exemple ceux qui ont fait **`pause`** `ck`, vont alors émettre le signal `has_next` pour demander l'exécution d'un autre instant de cette horloge.
- Le domaine décide ensuite s'il doit attendre ou pas le prochain instant de son horloge parente (lignes 15-16). C'est le cas si le compteur d'instants `cpt` a atteint la valeur maximale

@ exemples/annA/domain.rml

```
let mk_domain name parent period =
  let ck = mk_clock name in
  let cpt = ref 0 in (* compteur du nombre d'instants *)
  let process _domain =
    do
      map_option (fun pck -> emit pck.alive 1) parent;
      loop
        (* instant local *)
        emit ck.step; incr cpt; pause;
        await ck.alive(0) in
        (* fin de l'instant local *)
        emit ck.eoi; pause;
        pause;
        (* attente automatique *)
        if parent <> None &&
           (equal_option !cpt period or not (pre ck.has_next)) then (
          cpt := 0;
          match parent with
          | None -> ()
          | Some pck ->
              (* attente du prochain instant de l'horloge parente *)
              emit pck.alive(-1);
              run (await_step pck);
              map_option (fun pck -> emit pck.alive 1) parent
         )
      end
      ||
      run (sustain ck.alive)
      ||
      run (hold_server ck.hold)
    until ck.finished done;
    map_option (fun pck -> emit pck.alive (-1)) parent
  in
  _domain, ck
```

FIGURE A.2 – Encodage d'un domaine réactif

donnée en argument après le mot-clé **by** ou si aucun processus à l'intérieur n'a demandé de nouvel instant local, c'est-à-dire si `has_next` n'a pas été émis.

– Il recommence ensuite un nouvel instant local, en ayant auparavant attendu le prochain instant de son horloge parente si nécessaire (ligne 23).

L'encodage de la construction **pause** `ck` permet de clarifier la façon dont le domaine utilise le signal `has_next` pour implémenter l'attente automatique de son domaine parent :

```
let process pause_ck d =
  await d.eoi;
  emit d.has_next;
  await d.step
```

On voit que cette construction attend la fin de l'instant de l'horloge `ck`, puis émet le signal `has_next` pour signifier qu'il reste un processus à exécuter à l'instant suivant, puis enfin attend le début de ce prochain instant.

@ exemples/annA/domain.rml

```
let mk_signal d g ck =
  signal s_await default d gather g in
  signal s_emit in
  let process hold =
    (* cas special pour le premier instant *)
    if pre s_emit then
      (List.iter (fun x -> emit s_await x) (pre? s_emit))
    ||
    loop
      await s_emit;
      begin
        do
          run (sustain_list s_emit)
        until ck.step done
        ||
        await immediate ck.eoi;
        (* mise a jour de s_await *)
        List.iter (fun x -> emit s_await x) (pre? s_emit)
      end
    end
  in
  let s = { s_await = s_await; s_emit = s_emit; s_clock = ck } in
  emit ck.hold hold;
  s
```

FIGURE A.3 – Encodage d'un signal

Signaux

Chaque signal est traduit en un enregistrement contenant :
– L'horloge du signal.
– Un signal `s_emit` qui indique la présence du signal.
– Un signal `s_await` qui permet d'obtenir la valeur du signal.
L'émission d'une valeur sur un signal consiste à émettre cette valeur sur le signal `s_emit` :

```
let emit_v s v =
  emit s.s_emit v
```

L'attente d'un signal se fait par contre en utilisant le signal `s_await`. Il faut également signaler au domaine réactif correspondant qu'un processus souhaite s'exécuter dans le prochain instant local :

```
let process await_all s act_ck =
  await s.s_await(v) in
  emit act_ck.has_next;
  run (await_step act_ck);
  v
```

Le processus `await_all` termine bien à l'instant suivant de l'horloge du signal, puisque le signal `s_await` n'est émis que pendant la fin de l'instant de cette horloge. De la même manière, on lit la dernière valeur du signal en lisant la dernière valeur du signal `s_await`.

Chaque signal est accompagné d'un processus qui maintient `s_emit` entre son émission et la fin de l'instant de l'horloge du signal (lignes 12-14) et qui collecte les valeurs émises sur le signal pour calculer sa valeur. Pour cela, les valeurs émises sur le signal sont émises sur le signal `s_emit`, dont la fonction de combinaison garde la liste des valeurs émises. Ensuite, à la fin de l'instant de l'horloge du signal, on émet toutes ces valeurs en un seul instant (ligne 18) sur le signal `s_await`

dont la fonction de combinaison est celle associée au signal que l'on veut encoder. La valeur de `s_await` à cet instant est donc bien celle que l'on obtient en collectant les valeurs émises pendant tout l'instant de ce domaine réactif, y compris les instants des domaines enfants.

On ne peut pas lancer le processus `hold` à l'endroit où le signal est défini. En effet, un signal REACTIVEML peut a priori être utilisé n'importe où dans le programme. On doit donc le maintenir jusqu'à la fin du programme, ou plus précisément jusqu'à la fin de l'exécution du domaine auquel il est attaché. On lance donc dans chaque domaine un processus nommé `hold_server` qui va maintenir la valeur des signaux du domaine (ligne 27 de la figure A.2). :

```
let rec process hold_server hold =
  await hold(l) in
  run iter_p (proc p -> run p) l
  ||
  run hold_server
```

Le processus reçoit sur le signal `add` les processus de maintien qu'il lance avant de se remettre en attente de la prochaine émission de `add`. On émet sur le signal `hold` attaché au domaine le processus de maintien d'un signal au moment de sa création (ligne 23 de la figure A.3).

Traduction

Il nous reste encore un petit peu de travail pour pouvoir exécuter le résultat de notre encodage. Il s'agit des règles permettant de traduire une expression dans la syntaxe concrète de REACTIVEML avec domaines en une expression utilisant notre encodage. Ces règles sont données sur la figure A.4. Pour la majorité des constructions, il s'agit simplement d'appeler le processus ou la fonction correspondante de l'encodage. On peut tout de même signaler plusieurs particularités :

- Dans la traduction d'un domaine réactif, on lance en parallèle le corps du domaine, qui doit d'abord attendre la première activation de l'horloge du domaine, et le processus qui gère le fonctionnement du domaine. Une fois que le corps a terminé son exécution, on émet le signal `ck.finished` pour terminer l'exécution de ce dernier.
- Chaque processus est désormais paramétré par ce qu'on va appeler son horloge d'activation, noté ici `act_ck`. Il s'agit de l'horloge locale à l'endroit où le processus est lancé, comme on peut le voir dans la traduction de **`run`**.
- La dernière ligne du tableau montre la traduction du processus principal du programme. On le lance simplement dans le domaine réactif d'horloge `global_ck`.

Ces règles de traduction sont implémentées dans le compilateur REACTIVEML fourni avec ce manuscrit. Il faut appeler le compilateur avec la commande :

```
> rpmlc -s main -runtime Rpml2rml a.rml
```

où `main` est le processus principal du programme. Cette commande génère un fichier `a_gen.rml` que l'on peut ensuite compiler avec le compilateur REACTIVEML classique, par exemple en utilisant `rmlbuild`, après avoir copié le fichier `domain.rml` dans le même dossier :

```
> rmlbuild a_gen.rml.native
```

Le fichier fichier `exemples/annA/domain_example.rml` est un exemple fourni avec le manuscrit. On peut le compiler par les commandes :

```
> cd exemples/annA
> make domain_example.rml_gen.native
```

qui se chargent d'effectuer les différentes étapes décrites ci-dessus.

`pause ck`	`run pause_ck ck`

`domain(ck) do p done`	`let _domain, ck = mk_domain parent period in`
	`run _domain`
	`||`
	`run await_step ck; p; emit ck.finished`

`signal s default d` `  gather g in e`	`let s = mk_signal d g act_ck in e`
`emit s v`	`emit_v s v`
`await s(v) in p`	`let v = run await_all s act_ck in p`
`do p until s done`	`do p until s.s_await done;` `run await_step act_ck`
`do p when s done`	`do p when s.s_emit done`
`process e`	`fun act_ck -> process e`
`run e`	`run e act_ck`

`let process main = p`	`let gck_domain, gck =`
	`  mk_domain "global_ck" None None`
	
	`let process main =`
	`  run gck_domain`
	`  ||`
	`  run await_step gck; p; emit gck.finished`

FIGURE A.4 – Traduction de REACTIVEML avec domaines en REACTIVEML

ANNEXE B

Optimisation mémoire modulaire dans un langage synchrone flot de données

Nous présentons dans ce chapitre des travaux annexes sur une optimisation mémoire pour les langages synchrones flot de données. Il s'agit d'un travail qui a commencé pendant mon stage de recherche [Pas10] et qui a ensuite mené à la publication d'un article [GGPP12] récompensé du prix du meilleur article de la conférence LCTES'12. Nous présentons l'article dans sa version originale en anglais. Nous avons corrigé une erreur dans la définition de la notion d'interférence (Définition 7) et adapté la mise en page pour l'uniformiser avec le reste du manuscrit.

Ce travail a été mené sur le langage HEPTAGON. Il s'agit d'un langage synchrone flot de données intégrant des automates de mode hiérarchiques [CPP05] ainsi que des tableaux et les itérateurs associés [Mor07]. Son compilateur a été développé initialement par Marc Pouzet [BCHP08], puis repris et étendu par Gwenaël Delaval, Léonard Gérard, Adrien Guatto et moi-même. Il intègre désormais des travaux récents sur la synthèse de contrôleur [DMR10], la génération de code VHDL [GP10] et l'utilisation de futurs [CGP12] pour la parallélisation du code.

Le but de ce travail est de minimiser la mémoire allouée et le nombre de copies effectuées dans le code séquentiel généré à partir d'un programme HEPTAGON. L'approche choisie est de combiner une optimisation mémoire au moment de la compilation avec un système d'annotations donnant plus de contrôle au programmeur. Cette optimisation, que l'on nomme *allocation mémoire*, est présentée sous forme d'un problème de coloriage de graphe, à la manière du problème classique de l'allocation de registres [CAC+81]. On montre dans cet article comment étendre cette approche au cas des flots avec horloges et des registres synchrones. On couple cette optimisation forcément fragile avec un système d'annotations permettant au programmeur de gérer le partage mémoire entre nœuds, en forçant les modifications en place. Ces annotations permettent également d'importer de façon sûre des fonctions externes qui modifient en place leur argument. La correction des annotations est vérifiée par un système de types *semi-linéaires* [Wad90].

L'optimisation des copies de tableaux est indispensable pour les programmes dans lesquels on manipule les tableaux pour du calcul. L'allocation mémoire a ainsi été utilisée avec succès pour les exemples de l'extension d'HEPTAGON avec des futurs [CGP12]. D'autres travaux préliminaires sur la génération de code efficace [Thi13], notamment pour GPU [Gel11], ont également permis de montrer l'efficacité de l'optimisation. Les annotations semi-linéaires ont également été utilisées dans les deux cas. Bien qu'elles aient à chaque fois permis d'obtenir le résultat attendu, c'est-à-dire l'absence de copies inutiles, leur écriture a toutefois posé de nombreux soucis et requis plusieurs essais. C'est sans doute le prix à payer pour obtenir un système d'annotations avec un contrôle fin de la mémoire mais avec des messages d'erreurs compréhensibles par l'utilisateur.

Abstract

The generation of efficient sequential code for synchronous data-flow languages raises two intertwined issues: control and memory optimization. While the former has been extensively studied, for instance in the compilation of LUSTRE and SIGNAL, the latter has only been addressed in a restricted manner. Yet, memory optimization becomes a pressing issue when arrays are added to such languages.

This article presents a two-level solution to the memory optimization problem. It combines a compile-time optimization algorithm, reminiscent of register allocation, paired with language annotations on the source given by the designer. Annotations express in-place modifications and control where allocation is performed. Moreover, they allow external functions performing *in-place* modifications to be safely imported. Soundness of annotations is guaranteed by a semilinear type system and additional scheduling constraints. A key feature is that annotations for well-typed programs do not change the semantics of the language: removing them may lead to less efficient code but will not alter the semantics.

The method has been implemented in a new compiler for a LUSTRE-like synchronous language extended with hierarchical automata and arrays. Experiments show that the proposed approach removes most of the unnecessary array copies, resulting in faster code that uses less memory.

B.1 Introduction

Synchronous data-flow languages [BCE+03] are widely used for the design and implementation of embedded systems. The generation of sequential imperative code was addressed more than twenty years ago in the early work on LUSTRE [CPHP87] and SIGNAL [BLGJ91] and is routinely used in industrial tools such as SCADE. Its principle is to generate a transition function that computes a synchronous step of the system, which is then infinitely repeated. For tools like SCADE, code generation is done modularly, producing a single transition function per stream function, independently of the calling contexts [BCHP08].

Two critical optimizations have to be performed during the generation of sequential code: *control structure optimization* and *memory optimization*. Control optimization tries to reduce useless code at every reaction according to the value of certain boolean variables. Several methods have been proposed, ranging from local optimizations performed modularly [BCHP08] to more aggressive but non-modular ones [HRR91]. In this paper, we focus on the memory optimization problem. It aims to minimize the allocated memory and the number of copy operations when computing a reaction. This becomes an important issue in production compilers like SCADE 6 due to the presence of functional iterators over large arrays [Mor07]. Because these arrays are semantically functional — if t_1 and t_2 are arrays, $t_1 + t_2$ denotes a third array and the update $t_1\{i \leftarrow e\}$ returns a fresh copy whose i-th element is equal to e — the direct translation into sequential code is untenable for performance reasons. Arrays must be shared, with in-place modifications and useless copies eliminated as much as possible. Unfortunately, methods like register reuse [HRR91] and iterator fusion [Mor07] treat the problem only partially and locally to a block [Pag10]. Recently, this problem was addressed by S. Abu-Mahmeed et al. [AMMB+09] and applied on the data-flow language LABVIEW, but without proposing an interprocedural solution which is essential for good performance.

Contribution and organization of the paper We address the problem in a different and more unified way by combining a static memory allocation algorithm together with language annotations. The memory allocation is presented as a graph coloring problem like the well-known problem of register allocation [CAC+81]. The main novelties are the extension to *clocked streams* and the handling of *synchronous registers*. As the optimization is necessarily fragile, it is coupled with language annotations that give the designer precise control over interprocedural memory sharing. These annotations also allow to safely import external functions performing in-place modifications on their arguments. These annotations do not change the semantics of programs, that is, removing them leads to the same behavior. The soundness of annotations is enforced by a semilinear type system [Wad90] and additional scheduling constraints.

```
node halfSum(x:int)=(sum:int)
var sum1, sum2, x1, x2 :int;
    half :bool;
let
  half = true fby (not half);
  (x1, x2) = split half x;
  sum1 = 0 fby (sum1 + x1);
  sum2 = 0 fby (sum2 + x2);
  sum = merge half sum1 sum2;
tel
```

(a) HEPTAGON code

half	*tt*	*ff*	*tt*	*ff*	*tt*	*ff*	*tt*	...
x	1	7	4	2	9	3	1	...
x1	1		4		9		1	...
sum1	0		1		5		14	...
x2		7		2		3		...
sum2		0		7		9		...
sum	0	0	1	7	5	9	14	...

(b) Chronogram

```
//hS synchronous registers
typedef struct {
  int sum1, sum2;
  bool half;
} hSMem;

//hS transition function
int hSStep(int x, hSMem* m){
  int x1, x2, sum;
  if (m->half) {
    x1 = x; //split
    sum = m->sum1; //merge
    //update registers
    m->sum1 = m->sum1 + x1;
  } else {
    x2 = x;
    sum = m->sum2;
    m->sum2 = m->sum2 + x2;
  }
  //update registers
  m->half = not m->half;
  return sum;
}

int main() {
  //register initialization
  hSMem m = {0, 0, true};
  while(true)  //simulation loop
    out(hSStep(&m, in()));
}
```

(c) Simplified C code

Figure B.1: The `halfSum` node and corresponding generated code

The method has been applied to a LUSTRE-like language, called HEPTAGON, which extends LUSTRE with hierarchical automata and arrays. The material presented here could nonetheless be adapted to similar languages such as SCADE and the discrete subset of SIMULINK.

This article presents the language and memory issues with examples in Section B.2. Memory allocation is considered in Section B.3. Language annotations are described in Section B.4 and the semilinear type system is formalized in Section B.5. The changes induced on code generation are presented in Section B.6 together with benchmarks. Future extensions and related work are respectively discussed in Section B.7 and Section B.8 and we conclude in Section B.9.

B.2 Problem Statement

We informally introduce synchronous data-flow languages with a simple example, and then illustrate the memory issues tackled in the paper.

A simple example with two exclusive blocks A program is made of a list of declarations of *nodes*, i.e., functions acting on streams. Each node is defined by a set of mutually recursive equations over streams. For example, consider the node `halfSum` given in Figure B.1 that takes an input stream `x` and return an output `sum`.

The first equation defines the stream `half`. **`fby`** is a unit delay initialized with a constant value. It returns its first input concatenated with its second input. Thus, the value of `half` is the alternating sequence $tt.ff.tt.ff\ldots$. In the remainder, a variable defined by an equation of the form se `fby` e will be called a *synchronous register* (to avoid confusion with registers in register allocation).

The **`split`** operator is used to filter the stream `x` according to the boolean stream `half` (it replaces the **`when`** operator of LUSTRE). `x1` (resp. `x2`) is the stream made of the values of `x` when `half` is true (resp. false). It is absent otherwise. The **`merge`** operator joins the complementary streams `sum1` and `sum2`: `sum` is equal to `sum1` (resp. `sum2`) when `half` is true (resp. false). The *clock* of e, written $clock(e)$, defines the instants when the value of e is present. Here, it is a boolean formula of the form [BCHP08]:

$$ck ::= \texttt{base} \mid ck\ \texttt{on}\ c$$
$$c ::= x \mid \texttt{not}\ x$$

where `base` stands for the base clock and is interpreted as the constant stream of true values, ck `on` c is true when ck is true and c is present and true. For example in Figure B.1a, $clock(\texttt{half}) = \texttt{base}$ and $clock(\texttt{x1}) = \texttt{base on half}$. This notion of clock is also important for our memory allocation algorithm.

Figure B.1c shows a simplified version of the sequential code generated from `halfSum` with a main simulation loop. Notice that synchronous registers require special care: their current value is the one computed during the previous activation. That is why their equation is set after the code computing the variables.

Control Optimization During code generation, an equation $x = e$ is translated into an assignment which is executed only when the clock of e is true. It is important for efficiency to merge computations activated by the same clocks and not generate a separate conditional for each equation. Without this optimization, `x1`, `x2`, `sum1`, `sum2` and `sum` would have used one conditional each. The general form of this transformation is the basis of the compilation of SIGNAL [ABG95].

Memory Optimization Let us consider a more complex example to illustrate the memory problems that arise when using arrays in a data-flow language. The auxiliary function `swap` takes two indices and an array of size `n` (a global constant) and returns the same array with values at the given indices swapped. The `shuffle` node sequentially applies an array of permutations to an internal array and then returns the value at a given index:

```
const n : int = 100
const m : int = 3
const t_0 : float^n = 0.0^n

node swap(i, j : int; t_in : float^n) = (t_out : float^n)
var t_tmp : float^n;
let
   t_tmp = [ t_in with [i] = t_in[>j<] ];
   t_out = [ t_tmp with [j] = t_in[>i<] ];
tel

node shuffle(i_arr, j_arr : int^m; q : int) = (v : float)
var t, t_prev : float^n;
let
  t_prev = t_0 fby t;
  t = fold<<m>> swap(i_arr, j_arr, t_prev);
  v = t[>q<];
tel
```

`float^n` is the type of arrays of `float` of size `n` and `0.0^n` is the literal array filled with `n` `0.0` values. `[t with [i] = e]` returns an array equal to `t` except for the element at index `i` which is set to `e`. The language aims at critical systems so no out-of-bounds error is permitted, `t[>i<]` is thus the clipped index array access, returning the element of `t` at index $\min(\max(\mathtt{i}, 0), \mathtt{n} - 1)$. The `fold` iterator successively applies the `swap` function to the elements of `i_arr` and `j_arr`,[1] using an accumulator whose initial value is `t_prev`, the previous value of `t` initialized to a constant `t_0`.[2]

As the language has a functional semantics, each operation on an array creates a new array. This choice is compatible with block-diagram syntax and the inherently concurrent nature of the language, but makes efficient implementation harder. In `swap`, a naive implementation would allocate new arrays for `t_in` and `t_tmp` and copy the whole array twice. A common optimization in synchronous languages is to store a variable together with its previous value [HRR91]. In `shuffle`, `t` and `t_prev` may thus be stored together removing one unnecessary copy. Finally, the calling convention states that inputs are passed by value, so `m` unnecessary copies are done in the `fold` which calls `m` instances of `swap`.

Memory allocation avoids all copies inside the `swap` and `shuffle` nodes, but keeps the copies induced by the calling convention. The optimal implementation, which allocates only one array for the synchronous register `t_prev` and updates it in-place, is achieved in Section B.4 by combining memory allocation with annotations.

B.3 Memory Allocation

The memory allocation algorithm described in this section is presented as a graph coloring problem like register allocation. Indeed, both problems have similar goals and constraints: to share local variables without changing the semantics of a program. The main novelties of our approach are the extension to clocked streams and the handling of synchronous registers.

We recall the general definition of interference [CAC+81]:

Definition 1 (Interference (general))**.** Two variables interfere if they cannot be stored in the same memory location.

This notion has to be adapted to the clocked data-flow setting. We first define live ranges and interference on streams elements, following the usual definitions. The resulting notion of interference cannot be computed statically, so we adapt the definitions to streams, using clocks and the properties of synchronous registers, in order to get an abstract and easily computable definition.

In the following, we assume that synchronous registers are isolated in equations of the form $x = v$ `fby` y by a normalization pass. In this case, x *denotes both the stream and the register* so that $is\_reg(x) = true$. We consider every equation of a program as an infinite set of equations, one for each step, defining instantaneous variables. We note $x = (x_i)_{i\in\mathbb{N}}$ for a stream and $eq = (eq_j)_{j\in\mathbb{N}}$ for an equation.

$$eq : x = e \Leftrightarrow \forall i \geq 0.\ eq_i : x_i = e_i$$

$$eq : x = v \texttt{ fby } e \Leftrightarrow \begin{cases} i = 0 & eq_0 : x_0 = v \\ i > 0 & eq_i : x_{i+1} = \texttt{if } clock(x)_i \texttt{ then } e_i \texttt{ else } x_i \end{cases}$$

At each step, the value of a stream is computed if its clock is active. A register is persistent, so its value remains the same even if its clock is not active.

We suppose given a *schedule*, noted $\preceq$, that is a total order on equations compatible with data dependencies. $eq \preceq eq'$ means that eq must be computed before eq'. We note $\prec$ the associated strict order. The rest of the paper does not depend on the precise schedule chosen. We now define the *live range* of a variable, used to compute interference. $\mathrm{def}(x_i)$ is the equation defining x_i and $\mathrm{use}(x_i)$ is the set of equations using x_i.

1. If f has type signature $\tau_1 \times ... \times \tau_p \times \tau \longrightarrow \tau$, then `fold`$\langle n \rangle$ f has type signature $\tau_1\hat{}n \times ... \times \tau_p\hat{}n \times \tau\hat{}n \longrightarrow \tau\hat{}n$. When `m` equals 2, $t =$ `swap(i_arr[1], j_arr[1], swap(i_arr[0], j_arr[0], t_prev))`.
2. All these operators exist in SCADE 6.

Definition 2 (Live range)**.** We say that x_i is *alive* in the equation eq_j, denoted $live(x_i, eq_j)$, if:

$$live(x_i, eq_j) \triangleq (def(x_i) \prec eq_j) \wedge (\exists eq' \in use(x_i).eq_j \preceq eq')$$

Intuitively, a variable is alive between its definition and its last use. In particular, the live range of a register spans over two steps, as its equation defines the value for the next step. Interference can now be defined on streams in almost the same way as in register allocation:

Definition 3 (Interference (dynamic))**.** Two streams x and y interfere if they are alive in the same instance of an equation:

$$\exists i, j, eq, k.\ live(x_i, eq_k) \wedge live(y_j, eq_k)$$

This definition of interference cannot be computed statically as it depends on the actual values of clocks. For instance, x is never used in the equation $y = \texttt{merge}\ c\ 0\ x$ if c is always false. However, we can compute a static approximation of the live range of a stream, valid at every step, by using its associated clock. Let $def(x)$ be the equation defining the stream x and $use(x)$ the set of equations where x appears on the right-hand side. These two notions are over-approximations of the definitions given on streams elements. In particular, if there exist i and j such that $eq_j \in use(x_i)$, then $eq \in use(x)$, but the converse might not be true, as in the previous example.

Definition 4 (Live range (static))**.** We say that a stream x is *alive* in eq, denoted $live\_s(x, eq)$, if:

$$live\_s(x, eq) \triangleq (def(x) \prec eq) \wedge (\exists eq' \in use(x).eq \preceq eq')$$

In the `shuffle` example, `t` is alive in the equations defining `t_prev` and `v`. The similarity with Definition 2 makes it easy to see that the live range of a stream is an approximation of the live range of its elements (i.e. $\exists i, k.\ live(x_i, eq_k) \Rightarrow live\_s(x, eq)$). We now define the inclusion of clocks and the notion of *disjoint clocks* that approximates the liveness information given by clocks:

Definition 5 (Inclusion of clocks)**.** A clock ck is included in ck', which is denoted $ck \sqsubseteq ck'$, if:

$$ck \sqsubseteq ck' \quad \Leftrightarrow \quad \forall i \in \mathbb{N}.\ ck_i = tt \Rightarrow ck'_i = tt$$

Definition 6 (Disjoint clocks)**.** Two disjoint clocks are never both true during the same step:

$$dis\_ck(ck_1, ck_2) \triangleq \begin{cases} ck_1 = \texttt{base} \vee ck_2 = \texttt{base} & \Rightarrow false \\ ck_1 = ck\ \texttt{on}\ c \wedge ck_2 = ck\ \texttt{on not}\ c & \Rightarrow true \\ ck_1 = ck\ \texttt{on}\ c \wedge ck_2 = ck'\ \texttt{on}\ c' & \Rightarrow dis\_ck(ck, ck') \end{cases}$$

For instance, in the `halfSum` example, streams `x1` and `x2` have disjoint clocks and the clock of `sum1` is included in the clock of `sum`, which is equal to `base`. As a stream is only computed when its clock is true, two variables with disjoint clocks are never both needed during the same step so they never interfere. The value of a synchronous register must be kept even when its clock is not true as it may be used in a future step. As a consequence, a register interferes with any variable whose clock is not included in the clock of the register. Using these two ideas, we define an approximate notion of interference:

Definition 7 (Interference (static))**.** We say that x and y statically interfere, noted $x \boxtimes y$, if one of them is a register and the clock of the other is not included in the clock of the register, or if they are alive in the same equation and do not have disjoint clocks. Formally:

$$\begin{aligned} x \boxtimes y \triangleq\ & (is\_reg(x) \wedge clock(y) \not\sqsubseteq clock(x)) \vee (is\_reg(y) \wedge clock(x) \not\sqsubseteq clock(y)) \\ & \vee (\exists eq.\ live\_s(x, eq) \wedge live\_s(y, eq) \wedge \neg dis\_ck(clock(x), clock(y))) \end{aligned}$$

In the example `swap`, streams `t_in`, `t_tmp` and `t_out` do not interfere as they are never both alive in the same equation. The inputs and outputs of a node can be addressed similarly by adding two pseudo-equations, that are used only to compute the interference graph and never actually appear in the generated code:

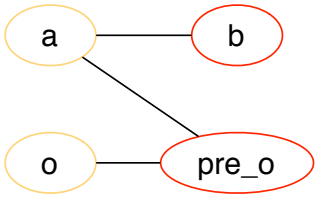
$$eq_{init} : a_1, \ldots, a_p = \texttt{read\_inputs}()$$
$$eq_{return} : \_ = \texttt{write\_outputs}(o_1, \ldots, o_q)$$

These equations state that inputs are alive at the beginning of node execution and that outputs are alive at the end of node execution.

Definition 8 (Interference graph)**.** An *interference graph* $G = (V, E, E_a)$ is an undirected graph where each vertex is associated with one stream and $(x, y) \in E$ if and only if $x \boxtimes y$.

In this example of an interference graph, the **map** operator applies a function to each element of an input array and returns an array of results. We suppose that the schedule $\preceq$ is the one given by the code on the left (i.e $eq_b \preceq eq_o \preceq eq_{pre\_o}$).

```
node p(a:float^n) = (o:float^n)
var pre_o, b:float^n;
let
  b = map<<n>>(-.)(a, pre_o);
  o = map<<n>>(+.)(a, b);
  pre_o = t_0 fby o;
tel
```



Normalization In order to maximize sharing opportunities, memory allocation is done after a normalization pass which creates new equations for temporary results. Indeed, in the `swap` example, `t_in` and `t_tmp` interfere as they are both used to define `t_out`. However, after the normalization, `t_in` and `t_tmp` no longer interfere:

```
v = t_in[>j<];
v_2 = t_in[>i<];
t_tmp = [ t_in with [i] = v ];
t_out = [ t_tmp with [j] = v_2 ];
```

Algorithm The algorithm to compute interferences closely follows the definitions. The list of live variables in each equation is computed using Definition 4, then the interference graphs (one for each type) are built using Definition 7. The DSATUR [Bré79] algorithm is used to color the graphs with a minimal number of colors. The result of the algorithm is a set of equivalence classes, where two variables in the same class have the same color and should be stored together.

Memory Allocation, Scheduling and Control Optimization We defined liveness according to a given schedule $\preceq$: the compiler performs memory allocation after the scheduling pass. The trade-off between scheduling and register allocation is a well-known problem in classic compilation (see [GH88] for instance). In our setting, the order of these passes is not an issue. Indeed, performing memory allocation before scheduling would be possible by considering only data-flow (or *true*) dependencies, but would always yield inferior results as any valid schedule respects these dependencies and thus induces strictly shorter live ranges.

However, in a clocked data-flow language, scheduling usually optimizes the control structure of the generated program by clustering equations with the same activation clock. This transformation may expand live ranges of streams, thereby limiting memory sharing. We modified the existing scheduling heuristic to favor memory optimization of arrays over control, by greedily minimizing the number of arrays alive at the same time. This heuristic is quadratic in the number of equations.

Memory allocation successfully avoids unnecessary copies within a node, for instance sharing `t_in`, `t_tmp` and `t_out` in `swap`. But the call-by-value convention states that inputs are passed by value, so one copy is made each time `swap` is called. These copies can be avoided if the input `t_in` is passed by reference and modified in-place in the generated code. This can be enforced by using the language annotations presented in the next section.

B.4 Language Annotations

Presentation

Location annotations enable the designer to express the in-place update of some inputs. If an input and an output are annotated with the same location, then the generated code will update the input in-place and return nothing.

Example In the example of Section B.2, the designer can express that `t_in` should be modified in-place by annotating `t_in`, `t_tmp` and `t_out` with the same location `r`. This is done using the notation `at r` beside the type declaration:

```
node swap(i, j:int; t_in:float^n at r) = (t_out:float^n at r)
var t_tmp : float^n at r;
let
   t_tmp = [ t_in with [i] = t_in[>j<] ];
   t_out = [ t_tmp with [j] = t_in[>i<] ];
tel
```

Only located variables can be given to a function that expects located arguments. To obtain a located variable `t_prev` from a non-located expression `t_0`, the programmer needs to explicitly initialize a new location `r` with the `init` construction:

```
node shuffle(i_arr, j_arr : int^m; q : int) = (v : float)
var t, t_prev : float^n at r;
let
  init t_prev = t_0 fby t;
  t = fold<<m>> swap(i_arr, j_arr, t_prev);
  v = t[>q<];
tel
```

As a result, the synchronous register `t_prev` will be updated in-place by `swap` and shared with `t`, so that no unwanted copy occurs.

The memory allocation algorithm is readily adapted to incorporate annotations: all variables with the same annotation are ensured to be stored in the same memory location (i.e., they correspond to the same vertex in the interference graph). However, the algorithm may still choose to share variables even if they are annotated with different locations.

Annotations may express that a function modifies its argument in-place even if it is not returned by the function, e.g.:

```
node f(mat:int^n^n at r) = (o:int)
```

which states that the body of `f` is allowed to overwrite `mat`.

Calling External Functions Location annotations are also used to safely import external functions that may modify their inputs in-place. For instance, we may import an efficient sorting function (e.g., written in C), with signature `void sort(int a[100])`, that modifies its input in-place. The usual way to achieve this is to add fake variables to enforce the necessary dependencies. Using location annotations, the function can safely be imported as:

```
fun sort(a : int^100 at r) = (o:int^100 at r)
```

Annotations *vs* side-effects In this proposal, we have chosen to maintain the block-diagram formalism, using annotations that can always be erased. Annotations are only used to control the efficiency of generated code. The semantics of a program with correct annotations remains the same if all annotations are removed. An alternative could have been to introduce mutable imperative variables, explicit side-effects and sequence in the source language, and take side-effects into account in the semantics.

Checking Annotations

Location annotations given by the programmer are unsound if two streams associated with the same location interfere. Annotated equations must satisfy well-formedness rules expressed as a type system and be statically schedulable.

We use a *semilinear* type system, following the work of Wadler in [Wad90]: *a value of semilinear type can be read multiple times and then updated once*. We call *update* an operator or function that deliberately modifies its argument in-place. For instance, physically modifying one element in an array (`[t with [i] = v]`) or calling the `swap` node are updates. A semilinear variable is defined either by updating a semilinear variable at the same location or by explicitly initializing a new location using the keyword **`init`**. The correctness of the annotations, that is, that two variables with the same semilinear type do not interfere, relies on the three following properties.

Property 1 (Init)**.** *A location is only initialized once.*

This is ensured by a simple syntactic check before typing that ensures that all the locations used in the inputs or with **`init`** are distinct.

Property 2 (Causality)**.** *If y results from an update of x, then the equation defining y is the last use of x with respect to $\preceq$.*

After its update, a semilinear variable cannot be read since its value has been overwritten, so it has to be dead (according to the schedule $\preceq$). The scheduling algorithm is modified in Section B.4 to enforce this property.

Property 3 (Type soundness)**.** *If two streams are associated with the same location, either one is obtained by successive updates from the other or they are obtained by updates from two variables defined by the application of the* `split` *operator.*

This property relies on the type system presented in Section B.5. In the first case, the streams are alive one after the other, while in the second, they have disjoint clocks. In both cases, they do not interfere. These three properties are essential to the correctness theorem for the system of annotations:

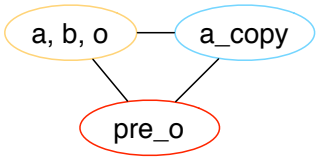
Theorem 4 (Annotation soundness)**.** *Two variables associated with the same location do not interfere.*

An annotated program is correct if it is well-typed (Property 3), schedulable (Property 2) and its locations are initialized only once (Property 1). A sketch of the proof of these properties and of Theorem 4 is given in Appendix B.B.

Scheduling

In order to ensure Property 2, static scheduling occurs after semilinear type checking. Extra dependencies between equations are added so that the update of a semilinear variable happens after all reads. This may introduce cycles, making scheduling impossible. For instance, in the node `p`, if variables `a`, `b` and `o` are annotated with the same location, the equation defining `o` reads `a`, so it should be scheduled before the equation defining `b` which updates `a` (i.e. $eq_o \prec eq_b$), but it also reads `b`, so it needs to be scheduled after the equation defining `b` (i.e. $eq_b \prec eq_o$). This can be fixed by introducing a copy of `a` (i.e., `a_copy = a`):

```
node p(a:float^n at r) = (o:float^n at r)
var b:float^n at r;
    a_copy, pre_o:float^n;
let
  a_copy = a;
  b = map<<n>>(-.)(a, pre_o);
  o = map<<n>>(+.)(a_copy, b);
  pre_o = t_0 fby o;
tel
```



The copy is not added automatically as we want to enforce the invariant that all streams at the same location are shared without any hidden copy.

B.5 Semilinear Type Checking

This section formalizes the semilinear type system that enforces the soundness property stated in Property 3. The system is used as a type checker, that is, with no type inference. For the sake of simplicity, we present a semilinear type system on a reduced version of the synchronous data-flow kernel used as an intermediate language during compilation.

Types

We define $\mathcal{R}_\top \triangleq \mathcal{R} \cup \{\top\}$, where $\mathcal{R}$ is the set of locations and $\top$ a special location representing the absence of information. We write by convention $r \in \mathcal{R}$ and $\rho \in \mathcal{R}_\top$. We denote by τ `at` r a semilinear type associated with the location r and by τ `at` $\top$ a non-linear type. Static expressions (se) are either values (v) or global constants (s). A plain type (τ) in the language is either a basic type or an array type. A type (μ) is given by a plain type and a location. We also define a node signature (σ):

$$se ::= v \mid s \qquad \tau ::= \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \tau\texttt{\^{}}se$$
$$\mu ::= \tau \texttt{ at } \rho \qquad \sigma ::= \forall r, \ldots, r.\mu^P \longrightarrow \mu^Q$$

An *update* is a function having an input and an output with the same semilinear type τ `at` r. We write $P \triangleq [1\,..\,p]$, $\mu^P \triangleq \mu_1 \times \cdots \times \mu_p$ and $x_P : \mu_P \triangleq x_1 : \mu_1, \ldots, x_p : \mu_p$ (likewise for any letter). We will also assume that $\mu_i \triangleq \tau_i$ `at` ρ_i.

Abstract Syntax

$$\begin{aligned}
n &::= \texttt{node } f(p;\ldots;p) = (p;\ldots;p) \texttt{ var } p,\ldots,p \texttt{ let } D \texttt{ tel}\\
eq &::= p = e \mid\ (p,\ldots,p) = f(w,\ldots,w) \mid p = se \texttt{ fby } w \mid (p,p) = \texttt{split } (x)\ x\\
&\quad \mid \texttt{init}\langle r\rangle\ p = se \texttt{ fby } w \mid \texttt{init}\langle r\rangle\ p = e\\
w &::= x \mid se\\
e &::= w \mid op(w,\ldots,w) \mid \texttt{merge } (x)\ w\ w\\
D &::= eq \mid D\ ;\ D\\
p &::= x : \mu
\end{aligned}$$

In this kernel, function arguments are extended values (w), either variables (x) or static expressions (se). A simple normalization pass can put any program into this form by introducing new local variables and equations. Although redundant, types also appear on the left-hand side of equations in order to simplify the presentation of the type system.

$$\textsc{Var}\ \frac{}{\Delta, \{x : \mu\} \vdash x : \mu} \qquad \textsc{Weakening}\ \frac{\Delta, \Gamma \vdash e : \mu'}{\Delta, \Gamma \uplus \{x : \mu\} \vdash e : \mu'}$$

$$\textsc{Copy}\ \frac{\Delta, \Gamma \uplus \{x : \tau\ \texttt{at}\ \top\} \uplus \{x : \tau\ \texttt{at}\ \top\} \vdash e : \mu}{\Delta, \Gamma \uplus \{x : \tau\ \texttt{at}\ \top\} \vdash e : \mu} \qquad \textsc{Linear Copy}\ \frac{\Delta, \Gamma \uplus \{y : \tau\ \texttt{at}\ r\} \uplus \{y : \tau\ \texttt{at}\ \top\} \vdash e : \mu}{\Delta, \Gamma \uplus \{y : \tau\ \texttt{at}\ r\} \vdash e : \mu}$$

(a) Linearity rules

$$\textsc{Const}\ \frac{\Delta \vdash se : \tau}{\Delta, \emptyset \vdash se : \tau\ \texttt{at}\ \top} \qquad \textsc{Block}\ \frac{\Delta, \Gamma \uplus \{x_L : \mu_L\} \vdash D}{\Delta, \Gamma \vdash \texttt{var}\ x_L : \mu_L\ \texttt{let}\ D\ \texttt{tel}} \qquad \textsc{EqList}\ \frac{\Delta, \Gamma \vdash D \qquad \Delta, \Gamma' \vdash D'}{\Delta, \Gamma \uplus \Gamma' \vdash D\ ;\ D'}$$

$$\textsc{Equation}\ \frac{\Delta, \Gamma \vdash e : \tau\ \texttt{at}\ \rho}{\Delta, \Gamma \vdash x : \tau\ \texttt{at}\ \rho = e} \qquad \textsc{Init}\ \frac{\Delta, \Gamma \vdash e : \tau\ \texttt{at}\ \top}{\Delta, \Gamma \vdash \texttt{init}\langle r\rangle\ x : \tau\ \texttt{at}\ r = e}$$

$$\textsc{Fby}\ \frac{\Delta, \Gamma \vdash w : \tau\ \texttt{at}\ \top \qquad \Delta, \emptyset \vdash se : \tau\ \texttt{at}\ \top}{\Delta, \Gamma \vdash x : \tau\ \texttt{at}\ \top = se\ \texttt{fby}\ w} \qquad \textsc{InitFby}\ \frac{\Delta, \Gamma \vdash w : \tau\ \texttt{at}\ r \qquad \Delta, \emptyset \vdash se : \tau\ \texttt{at}\ \top}{\Delta, \Gamma \vdash \texttt{init}\langle r\rangle\ x : \tau\ \texttt{at}\ r = se\ \texttt{fby}\ w}$$

$$\textsc{Merge}\ \frac{\Delta, \Gamma \vdash w_1 : \tau\ \texttt{at}\ \rho \qquad \Delta, \Gamma \vdash w_2 : \tau\ \texttt{at}\ \rho \qquad \Delta, \Gamma' \vdash x : \texttt{bool}}{\Delta, \Gamma \uplus \Gamma' \vdash\ \texttt{merge}\ (x)\ w_1\ w_2 : \tau\ \texttt{at}\ \rho} \qquad \textsc{Split}\ \frac{\Delta, \Gamma \vdash y : \mu \qquad \Delta, \Gamma' \vdash x : \texttt{bool}}{\Delta, \Gamma \uplus \Gamma' \vdash (y_1 : \mu, y_2 : \mu) = \texttt{split}\ (x)\ y}$$

(b) Expression and Equation rules

$$\textsc{App}\ \frac{\Delta \vdash f : \mu^P \longrightarrow \mu'^Q \qquad (\Delta, \Gamma_i \vdash w_i : \mu_i)_{i=1\ldots p}}{\Delta, \Gamma' \uplus \biguplus_{1\ldots p} \Gamma_i \vdash (x_Q : \mu'_Q) = f(w_1, \ldots, w_p)} \qquad \textsc{Node}\ \frac{\Delta, \{x_P : \mu_P\} \uplus \{x'_Q : \tau'_Q\ \texttt{at}\ \top\} \vdash b \qquad \sigma = gen(\mu^P \longrightarrow \mu'^Q) \qquad \texttt{well\_formed}(\sigma)}{\Delta \vdash \texttt{node}\ f(x_P : \mu_P) = (x'_Q : \mu'_Q)\ b : \sigma}$$

$$\textsc{Inst}\ \frac{\Delta(f) = \forall r_J.\mu^P \longrightarrow \mu'^Q \qquad \forall i, j \in J.\ r'_i = r'_j \Rightarrow i = j}{\Delta \vdash f : \mu^P \longrightarrow \mu'^Q\ [r_J \leftarrow r'_J]} \qquad \textsc{Gen}\ \frac{\{r'_i\}_{i=1\ldots j} = \{\rho_i \mid i = 1 \ldots p \wedge \rho_i \neq \top\}}{gen(\mu^P \longrightarrow \mu'^Q) = \forall r'_J.\mu^P \longrightarrow \mu'^Q}$$

$$\texttt{well\_formed}(\forall r_J.\mu^P \longrightarrow \mu'^Q) \triangleq \forall i, j, k. \begin{cases} \rho'_j \neq \top \Rightarrow \exists i.\ \mu_i = \mu'_j & \text{(WF1)} \\ \rho_i = \rho_k \neq \top \Rightarrow i = k & \text{(WF2)} \\ \rho'_j = \rho'_k \neq \top \Rightarrow j = k & \text{(WF3)} \end{cases}$$

(c) Function-related rules

Figure B.2: Semilinear Typing Rules

Typing Rules

The global and local typing environments, respectively written Δ and Γ are defined by ($\uplus$ stands for the union of multisets):

$$\Delta ::= \emptyset \mid \Delta \cup \{f : \sigma\} \mid \Delta \cup \{s : \tau\}$$
$$\Gamma ::= \emptyset \mid \Gamma \uplus \{x : \mu\}$$

The typing judgments are:

$$\Delta, \Gamma \vdash e : \mu \qquad \Delta \vdash se : \tau \qquad \Delta \vdash f : \sigma \qquad \Delta, \Gamma \vdash b \qquad \Delta, \Gamma \vdash D$$

which respectively mean that the expression e has type μ, the static expression se has type τ, the function f has signature σ and the block b or equations D are well-typed, in the global and local environments Δ and Γ.

The typing rules are given in Figure B.2. The size of some rules comes from the presence of n-ary functions returning multiple values, as in most block diagram languages. The most important rules are the ones that express the linearity properties (Figure B.2a).

1. The VAR rule is common to all linear type systems: it shows that each occurrence of x in the source corresponds to one and only one occurrence of x in the local environment, which is a multiset.
2. WEAKENING allows the removal of unnecessary elements from the environment.
3. The COPY and LINEAR COPY rules show the difference between semilinear and non-linear variables. For a non-linear variable x (of type τ `at` $\top$), we can duplicate its occurrence in the environment as much as we want, in order to use them as arguments for multiple reads. Conversely, the semilinear occurrence of a semilinear variable y (of type τ `at` r), cannot be duplicated. It is used in the typing rule of its only update. We can nevertheless create other occurrences of the same y with a non-linear type, in order to use them for multiple reads.
4. There are two ways to define a semilinear variable. The first one is to apply an update to another variable of the same type with the EQUATION rule. This is the case for instance for `t_tmp` and `t_out` in the `swap` example. The second one consists in initializing a new location from a non-linear variable with INIT.
5. The INITFBY rule ensures a correct use of semilinear synchronous registers. As two synchronous registers should always interfere, they can never have the same semilinear type. Except for the presence of `init`$\langle r \rangle$, this rule is the same as the application of an update, writing the value used in the next instant. The presence of `init` attests that the location r is initialized by the register with the value of the previous instant. Looking back at the `shuffle` example, it is clear that, in order to be able to modify the synchronous register `t_prev` in-place, it has to be defined as an update of `t`, which is itself an update of the previous value of `t_prev`. The location `r` is initialized at the first instant by `t_0`.
6. The MERGE rule uses a single local environment to type its arguments (unlike APP for instance) as we know that they have disjoint clocks. The SPLIT rule creates two variables with the same semilinear type, but it is safe since they have disjoint clocks.
7. The EQLIST rule conforms to the equational nature of our data-flow kernel: equation ordering does not matter and the type system is thus independent from scheduling.
8. The NODE rule states the constraints that a node signature must respect. Locations used in the inputs (resp. the outputs) must be distinct from each other (WF2) (resp. (WF3)). A node application cannot create a location: locations appearing in the outputs must appear in the inputs (WF1). Semilinear outputs can only be read (and not updated) as their value is needed at the end of the step, so they are added to the local environment with a non-linear type τ'_Q `at` $\top$.

Array Operators Semilinear typing extends to array operators ($\gamma \leq \rho$ stands for $\rho \neq \top \Rightarrow \gamma = \rho$):

$$\textsc{ArrayUpdate}\quad \frac{\Delta, \Gamma \vdash w : \tau\hat{}n \texttt{ at } \rho \qquad \Delta, \Gamma_1 \vdash w_1 : \texttt{int} \qquad \Delta, \Gamma_2 \vdash w_2 : \tau \texttt{ at } \top}{\Delta, \Gamma \uplus \Gamma_1 \uplus \Gamma_2 \vdash x : \tau\hat{}n \texttt{ at } \rho = [w \texttt{ with } [w_1] = w_2]}$$

$$\textsc{Map}\quad \frac{\begin{array}{c}\Delta \vdash f : (\tau_i \texttt{ at } \rho_i)^P \longrightarrow (\tau'_j \texttt{ at } \rho'_j)^Q \\ \sigma = (\tau_i\hat{}n \texttt{ at } \gamma_i)^P \longrightarrow (\tau'_j\hat{}n \texttt{ at } \gamma'_j)^Q \qquad \texttt{well\_formed}(\sigma) \qquad \forall i.\ \gamma_i \leq \rho_i \qquad \forall j.\ \gamma'_j \leq \rho'_j\end{array}}{\Delta \vdash \texttt{map}\langle n \rangle\ f : \sigma}$$

$$\textsc{Fold}\quad \frac{\Delta \vdash f : \tau_1 \texttt{ at } \top \times \ldots \times \tau_p \texttt{ at } \top \times \tau \texttt{ at } \rho \longrightarrow \tau \texttt{ at } \rho \qquad \gamma \leq \rho}{\Delta \vdash \texttt{fold}\langle n \rangle\ f : \tau_1\hat{}n \texttt{ at } \top \times \ldots \times \tau_p\hat{}n \texttt{ at } \top \times \tau \texttt{ at } \gamma \longrightarrow \tau \texttt{ at } \gamma}$$

The ARRAYUPDATE rule shows that modifying one element of an array can either be done in-place for semilinear variables (if $\rho \neq \top$) or possibly with a copy for other variables ($\rho = \top$).

The MAP and FOLD rules state all the possible signatures according to f. Map applies f to each element of its input arrays, so we can modify them in-place. However, if f modifies one of its inputs in-place, the corresponding array has to be modified in-place. Fold iterates f over the accumulator (the last argument), which may be modified in-place. It has to if f requires it. The other arguments are only read.

B.6 Implementation and Experiments

The material presented here on a kernel language has been implemented in the compiler of a richer synchronous language called HEPTAGON. The language allows the mixing of data-flow equations with hierarchical automata [CPP05]. Automata are eliminated by a source-to-source translation into the data-flow kernel. The language also supports a comprehensive set of operators on arrays [Mor07] and a simple form of parametricity compiled to C by macro-expansion. Apart from memory allocation, it implements two traditional optimizations for synchronous data-flow programs: iterator fusion [Mor07] and data-flow minimization. [3] The type checker is implemented in a simple, syntax-directed manner.

Sequential Code Generation

Following [BCHP08], the data-flow kernel is first translated into a small imperative intermediate language called OBC. The translation from OBC to existing sequential languages is then straightforward. Backends for C and JAVA have been implemented.

In OBC, the transition function of a node f is encapsulated with its internal state which stores the values of the synchronous registers from f. This encapsulation, called **`machine`**, is made of a list of state variables (declared with the **`registers`** keyword), a list of instances of other machines used by the machine (introduced by **`instances`**) and a **`step`** method for the transition function. The body of the transition function is expressed in a simple imperative language. Figure B.3a (respectively B.3c) shows the OBC code (respectively C code) corresponding to the translation of node `shuffle`, without memory optimization.

Expressing sharing in the intermediate sequential code In the original version of OBC [BCHP08], programs were forced to be in *Static Single Assignment* (SSA) form with all arguments of a method passed by value. In order to be able to share a location, we added *mutable* variables that can be assigned multiple times and *mutable* inputs that are passed by reference.

3. Data-flow minimization generalizes Common Subexpression Elimination. E.g., equations $x = 1\,\texttt{fby}\,x + 1$ and $y = 1\,\texttt{fby}\,y + 1$ reduce to a single one.

```
const n = 100
const m = 3
const t_0 = 0.0^n

machine shuffle =
  registers t_prev:float^n = t_0;
  instances swap:swap[m];

  step(i_arr, j_arr:int^m; q:int) = (v:float) {
    var t:float^n;
    t = this.t_prev;
    for i = 0 to m-1 do
       t = swap[i].step(i_arr[i], j_arr[i], t);
    this.t_prev = t;
    v = t[between(q, n)];
  }
```

(a) OBC without memory optimization

```
machine swap =
  step(i, j:int; mutable t_in:float^n) = () {
    var v_2, v:float;
    v = t_in[between(i,n)];
    v_2 = t_in[between(j,n)];
    if (i<n && i<=0) t_in[i] = v_2;
    if (j<n && 0<=j) t_in[j] = v;
  }

machine shuffle =
  registers mutable t_prev: float^n = t_0;
  instances swap: swap[m];

  step(i_arr, j_arr:int^m, q:int) = (v: float) {
    for i = 0 to m-1 do
      swap[i].step(i_arr[i], j_arr[i], this.t_prev);
    v = this.t_prev[between(q,n)];
  }
```

(b) OBC with memory optimization

```
#define between(idx, n)\
  (((idx)>=(n))?(n)-1:((idx)<0?0:(idx)))
static const int n = 100;
static const int m = 3;
struct shuffle_mem {float t_prev[100];};
struct swap_out {float t_out[100];};

float shuffle_step(const int i_arr[3], const int j_arr[3],
            int q, struct shuffle_mem* this) {
  float t[100];
  struct swap_out out;
  for (int i_1 = 0; i_1 < n; ++i_1)
    t[i_1] = this->t_prev[i_1];
  for (int i = 0; i < m; ++i) {
    swap_step(i_arr[i],j_arr[i],t,&out);
    for (int i_2 = 0; i_2 < n; ++i_2)
      t[i_2] = out.t_out[i_2];
  }
  for (int i_3 = 0; i_3 < n; ++i_3)
    this->t_prev[i_3] = t[i_3];
  return t[between(q, n)];
}
```

(c) C code without memory optimization

```
struct shuffle_mem {float t_prev[100];};

void swap_step(int i, int j, float t_in[100]) {
  float v_2, v;
  v = t_in[between(i, n)];
  v_2 = t_in[between(j, n)];
  if (i<n && 0<=i) t_in[i] = v_2;
  if (j<n && 0<=j) t_in[j] = v;
}

void shuffle_init(struct shuffle_mem* this) {
  for (int i = 0; i < n; ++i)
    this->t_prev[i] = 0.000000;
}

float shuffle_step(const int i_arr[3], const int j_arr[3],
            int q, struct shuffle_mem* this) {
  for (int i = 0; i < m; ++i)
    swap_step(i_arr[i], j_arr[i], this->t_prev);
  return this->t_prev[between(q, n)];
}
```

(d) C with memory optimization

Figure B.3: A node and the corresponding generated code

The result of the memory allocation described in Section B.3 is a set of equivalence classes, where two variables in the same class must be stored together. Sharing is applied by a modular source-to-source transformation in OBC. A representative is chosen in each equivalence class (either an input or synchronous register if there is one, otherwise any variable). All other variables in the equivalence class are replaced by this representative and unused variables are removed. An input shared with an output becomes mutable (to express the in-place modification) and the output is removed. Finally, node calls have to take into account the removed outputs. For instance, in the `shuffle` node, `t_next` is chosen as the representative for `t` and `t_next`, and it is passed by reference to `swap`, that does not return anything after the transformation. Figures B.3a and B.3b show the OBC code before and after the transformation (the `swap` node without optimization is in Appendix B.A). In the end, all the updates are performed in-place in the synchronous register.

Experiments

The graphs in Figure B.4 show both the effects of memory optimization alone and combined with annotations on the generated step function. The figures are given relatively to the unoptimized

results. We use the CompCert [Ler09] 1.9.1 C compiler to generate PowerPC code, and compute worst-case execution times (WCET) with the Open Tool for Adaptative WCET Analysis. [4]

As the `shuffle` example showed, annotations are essential as many unnecessary copies are made when iterating over arrays. Thanks to them, the generated code performs no array copies and is thus much faster with memory optimization. The program is tested with an array of size 50.

The second example sorts an array of size n in n^2 steps by swapping two elements at each step. Here, although the coloring done by memory allocation is optimal in terms of the number of colors, i.e., in terms of memory used (as seen in the second graph), it awkwardly shares arrays. Annotations are used to force one coloring which removes one unnecessary array copy.

The third example is a simplified version of a radar control panel (about 1 kLOC), adapted from one of SCADE demos. [5] Even though the program uses only small arrays (of size 2 to 6) and records, the use of annotations still results in performance improvements.

The last example is a simple downscaling image filter. It mainly consists of repeated vector-style computations on pixels, represented as floating-point arrays of size 4. Here, annotations give a small time boost, and provide negligible improvements in memory occupancy.

Note that the optimization is performed both on structured and scalar variables, but that the impact of the latter on execution times and memory use are negligible. However, the generated code is shorter and more readable, both in terms of instruction and variable counts. The last example, composed of multiple nested automata typical of the industrial use of SCADE, illustrates this point. As the program is composed of one complex monolithic node, our annotations did not give any extra benefits.

B.7 Discussion

Semilinear typing A more natural approach to annotations would have been to treat location annotations as coloring instructions for the interference graph, without any constraints, and report an error if coloring fails. While it may appear simpler, the drawback is that fixing incorrect annotations would require an understanding of the interferences and the choices made by the scheduler. On the contrary, the type system we have considered is independent from memory allocation. It is useful even without memory allocation, e.g., to manually express in-place modifications or import external functions with side-effects.

Extensions Most of the additional features of the HEPTAGON language are implemented by source-to-source transformations to a data-flow kernel close to the one presented in Section B.5. Performing memory allocation on this kernel is simpler while retaining all possibilities for optimization. This is not surprising since this kernel shares much similarity with the SSA form used in compilers such as GCC [Nov03]. The only change needed is a simple extension of the notion of disjoint clocks to share synchronous registers between states of an automaton separated by a reset. The compiler is also able to share fields inside a record, by treating them individually in the interference graph. The changes required to adapt the semilinear type system to the full language are also minimal.

B.8 Related Work

The problem of eliminating array copies, also known as the *aggregate update problem* [HB85], is shared by all functional languages. As a consequence, many solutions to this problem have been proposed. They can be divided into three families. The first relies on data structures at run time, such as a *garbage collector* or *reference counting*. In the special case of arrays, one can use *persistent arrays* [Die89], where only modifications to an array are stored instead of copying the whole array.

The second family of solutions tries to to tackle the problem at compile time. Static analyses try to find the last use of variables, to know when an update can safely be done in-place. This

4. The tool is available at `http://otawa.fr`.
5. The Mission Computer demo is available at: `http://www.esterel-technologies.com/technology/demos`

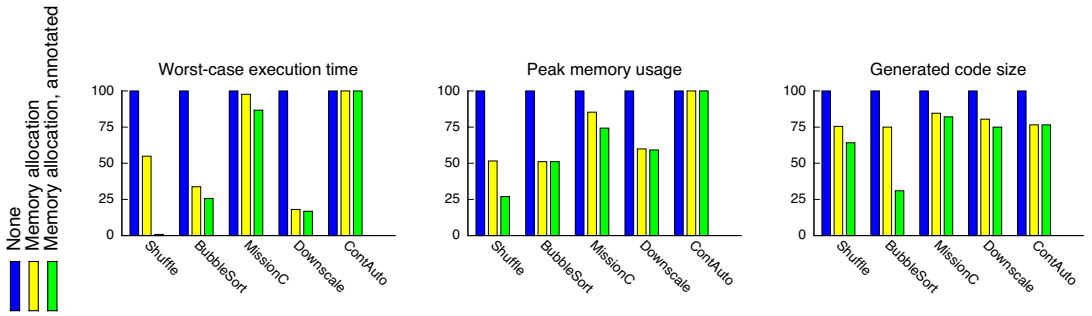


Figure B.4: Experimental results: worst-case execution time, maximum memory use and generated code size (lower is better)

information on the live-range of variables can be found using heuristics [SGH93], abstract interpretation [Ode91] or through Hindley-Milner type inference [Bak90]. All these methods are coupled with a dynamic system such as reference counting to deal with cases that cannot be decided statically. Another related method is *deforestation* [Wad88], which eliminates temporary data structures by transforming the code. All these static optimizations are fragile and do not allow direct control on memory sharing, as it is possible with explicit annotations.

The third family of solutions uses type systems to only accept programs where memory can be reused. They are based on *linear logic* [Gir87]: a variable of linear type can only be used once, and can thus be updated in-place. This restriction of a single use is too strong to be used in a real programming language. Many proposals have been made to relax it whilst maintaining strong enough invariants to enable memory sharing. One solution is to syntactically limit a scope where a linear variable can be considered as non-linear [Wad90], or to mix linear and non-linear types in the language, as in *uniqueness* typing [BS95]. The type system presented here is based on the same principles, the main novelty being the use of locations, which is made necessary by the presence of n-ary functions.

The choices made here, in particular in terms of calling convention, are similar to those made in [SGH93], although our formalism is more generic. The closest work is that of S. Abu-Mahmeed et al. [AMMB+09] and applied to LABVIEW. They propose a greedy algorithm that chooses successively, using a notion of cost, an operation to do in-place until dependencies make it impossible to choose another one. Our approach is more general in that it not only focuses on in-place modifications but that it can also share unrelated variables. In addition, we also propose a solution to interprocedural memory optimization. However, their notion of cost could be used to improve our greedy scheduling algorithm.

In the field of data-flow synchronous languages, a classic memory optimization consists in storing $\texttt{pre}\,x$ and x in the same memory location [HRR91]. This can be done if all the reads of $\texttt{pre}\,x$ occur before the definition of x. In our formalism, it implies that x and $\texttt{pre}\,x$ do not interfere, so this optimization is a particular case of the more general approach presented here.

B.9 Conclusion

This paper has presented a method for optimizing memory when compiling synchronous data-flow programs to sequential code. The method combines a static memory allocation algorithm with explicit language annotations. Memory allocation is expressed as a graph coloring problem, which links it to the classic register allocation problem. The soundness of annotations is checked by a semilinear type system and additional scheduling constraints. This ensures that annotations do not change the original functional semantics of the language but only its efficient code generation. A possible extension is the automatic inference of the annotations.

B.A swap without memory optimization

```
machine swap =
  step(i: int, j: int, t_in: float^n) = (t_out: float^n) {
    var v_2: float; v: float; t_tmp: float^n;
    v_2 = t_in[between(j, n)];
    v = t_in[between(i, n)];
    if (i<n && 0<=i) {
      for i_4 = 0 to i-1 do
        t_tmp[i_4] = t_in[i_4]
      t_tmp[i] = v_2;
      for i_5 = i+1 to n-1 do
        t_tmp[i_5] = t_in[i_5]
    } else {
      t_tmp = t_in
    }
    if (j<n && 0<=j) {
      for i_2 = 0 to j-1 do
        t_out[i_2] = t_tmp[i_2]
      t_out[j] = v;
      for i_3 = j+1 to n-1 do
        t_out[i_3] = t_tmp[i_3]
    } else {
      t_out = t_tmp
    }
  }
```

B.B Proof of correctness of the annotation system

Definition 9 (Root)**.** We call x the *root* of location r if $x : \tau$ `at` r and x is either an input or a variable defined by $\texttt{init}\langle r\rangle\ x = e$. We denote $\texttt{root}(r) = x$.

Property 5 (Init)**.** *There is only one root for each location r.*

Proof. By definition of `init`. □

Property 6. *If x is semilinear and is a register, then x is a root.*

Proof. See INITFBY rule. □

Definition 10 (Update relation)**.** We say that x is an *update* of y, denoted $y \rhd x$, if $x, y : \tau$ `at` r and one of the following applies :

- $(x_1, \ldots, x_q) = f(w_1, \ldots, w_p)$ with $f : (\tau_i \texttt{ at } \rho_i)^P \longrightarrow (\tau'_j \texttt{ at } \rho'_j)^Q$, $x_i = x$, $w_j = y$ and $r_i = r'_j$.
- $x = y$
- $x = \texttt{merge}\ (c)\ w_1\ w_2$ with $w_j = y$
- $(x_1 : \mu, x_2 : \mu) = \texttt{split}\ (c)\ y$ with $x_i = x$

It should be noted that there is no case corresponding to the INITFBY rule.

Definition 11 (Update order)**.** We define the (partial) order $\trianglerighteq$ as the smallest reflexive transitive antisymmetric relation such that $y \rhd x \Rightarrow y \trianglerighteq x$.

Property 7. *If x is semilinear, then x is either a root or there exists a y such that x is an update of y.*

Proof. By induction on the typing rules. □

Property 8. *If $x : \tau$ `at` r then $\texttt{root}(r) \trianglerighteq x$.*

Property 9 (Type soundness)**.** *If two variables are associated with the same location, either one is obtained by successive updates from the other or they are obtained by updates from two variables resulting of a* `split`*. Formally, if* $x, y : \tau$ `at` r *then one of the following condition is true:*

- $y \trianglerighteq x$ *(resp.* $x \trianglerighteq y$*).*
- *There exists* $z, z_x, z_y : \tau$ `at` r *such that* $z \trianglerighteq x$*,* $z \trianglerighteq y$*,* $z_x \trianglerighteq x$*,* $z_y \trianglerighteq y$*,* $z_x \not\trianglerighteq y$*,* $z_y \not\trianglerighteq x$ *and* (z_x, z_y) = `split` (c) z *(or the opposite).*

Proof. If $y \trianglerighteq x$ (resp $x \trianglerighteq y$), then y (resp x) is the maximum we are looking for. Otherwise, let's denote $S = \{z \mid z \trianglerighteq x \wedge z \trianglerighteq y\}$. We know that $\texttt{root}(r) \in S$. There exists z such that $\texttt{root}(r) \triangleright z$. If z is still in S, we can iterate with z. Eventually, we find z_0 such that $z_0 \not\trianglerighteq x$ and $z_0 \trianglerighteq y$ (or reciprocally) (we know that we have not encountered x or y, otherwise we would have been in one of the first two cases). We also know that there exists $z_x, z_y : \tau$ `at` r such that $z_x \trianglerighteq x$ and $z_y \trianglerighteq y$ and (z_x, z_y) = `split` (c) z_0 (or the opposite), as this is the only case where two variables can be updates of a variable. □

Property 10 (Causality)**.** *If* x *is an update of* y*, then the equation defining* x *is the last use of* y*:*

$$y \triangleright x \Rightarrow \forall eq \in \mathrm{use}(y).\ eq \preceq \mathrm{def}(x)$$

Proof. If $y \triangleright x$, then $\mathrm{def}(x)$ is the only update of y and its last use, as guaranteed by the modified scheduling algorithm (see Section B.4). □

Property 11. *If* $y \trianglerighteq x$ *then* x *and* y *do not interfere.*

Proof. If $y \trianglerighteq x$ then there exists $y_1, \ldots, y_m$ such that $y \triangleright y_1 \ldots \triangleright y_m \triangleright x$.
Then $\forall eq \in \mathrm{use}(y).\ \mathrm{use}(y).\ eq \preceq \mathrm{def}(y_1) \preceq \mathrm{use}(y_1) \preceq \ldots \preceq \mathrm{def}(x)$ by applying m times Property 10 and by transitivity of $\preceq$. It means that $\forall eq.\mathrm{live\_s}(y, eq) \Rightarrow \neg\mathrm{live\_s}(x, eq)$, so x and y do not interfere. □

Theorem 12 (Annotation soundness)**.** *Two variables associated with the same location do not interfere:*

$$\exists \tau, r.\ x, y : \tau \texttt{ at } r \Rightarrow \neg\,(x \boxtimes y)$$

Proof. Let $x, y : \tau$ `at` r.

Case 1 x and y are registers.
This is impossible as a semilinear register is the unique root of its location (Property 5 and 6).

Case 2 y is a register, x is not.
Then y is the root of location r so $y \trianglerighteq x$. By Property 11, we have that x and y do not interfere.

Case 3 x and y are not registers.

- Either $y \trianglerighteq x$ (or reciprocally) then x and y do not interfere (Property 11).
- Or there exists z, z_x, z_y such that $z, z_x \trianglerighteq x$ and $z, z_y \trianglerighteq y$ and (z_x, z_y) = `split` c z by Property 3. Then we can show that x (resp. y) is on the same clock or a slower clock than z_x (resp. z_y), which proves that x and y have disjoint clocks. As they are not registers, it follows that x and y do not interfere.

□

Bibliographie

[ABG95] T. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the Data-flow Synchronous Language SIGNAL. In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

[ABMS11] P. Attar, F Boussinot, L. Mandel, and J.-F. Susini. Proposal for a Dynamic Synchronous Language. PARTOUT, 2011.

[AD07] R.M. Amadio and F. Dabrowski. Feasible reactivity in a synchronous π-calculus. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 221–230. ACM, 2007.

[AM13] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP'13*, 2013.

[AMMB+09] S. Abu-Mahmeed, C. Mccosh, Z. Budimlić, K. Kennedy, K. Ravindran, K. Hogan, P. Austin, S. Rogers, and J. Kornerup. Scheduling tasks to maximize usage of aggregate variables in place. In *CC '09 : Proceedings of the 18th International Conference on Compiler Construction*, pages 204–219, Berlin, Heidelberg, 2009. Springer-Verlag.

[ANN99] T. Amtoft, F. Nielson, and H. Nielson. *Type and Effect Systems : Behaviours for Concurrency*. Imperial College Press, 1999.

[AP13] A. Abel and B. Pientka. Well-founded recursion with copatterns. In *ICFP'13*, 2013.

[Bak90] H. G. Baker. Unify and conquer. In *LFP '90 : Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 218–226, New York, NY, USA, 1990. ACM.

[Bal97] F. Balarin. *Hardware-software co-design of embedded systems : the POLIS approach*. Springer, 1997.

[BC84] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, pages 389–448, 1984.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[BCHP08] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES '08 : Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 121–130, New York, NY, USA, 2008. ACM.

[BD08] F. Boussinot and F. Dabrowski. Safe Reactive Programming : The FunLoft Proposal. In *Proc. of MULTIPROG – First Workshop on Programmability Issues for Multi-Core Computers*, 2008.

[Ber92] G. Berry. Esterel on hardware, mechanized reasoning and hardware design, 1992.

[Ber96] G. Berry. The constructive semantics of pure Esterel, 1996. http://www-sop.inria.fr/members/Gerard.Berry/.

[Ber97] G. Berry. The Esterel v5 language primer. *Ecole des Mines and INRIA*, 1997. http://www-sop.inria.fr/members/Gerard.Berry/.

[BFG+04] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in computer science*, 14(01) :97–141, 2004.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[BHLM94] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogeneous systems. 1994.

[BJMP13] G. Baudart, F. Jacquemard, L. Mandel, and M. Pouzet. A synchronous embedding of Antescofo, a domain-specific language for interactive mixed music. In *Thirteen International Conference on Embedded Software (EMSOFT'13)*, Montreal, Canada, September 2013.

[BLGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations : the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2) :103 – 149, 1991.

[BNS11] G. Berry, C. Nicolas, and M. Serrano. Hiphop : a synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, pages 49–56. ACM, 2011.

[Bou91] F. Boussinot. Reactive C : an extension of C to program reactive systems. *Software : Practice and Experience*, 21(4) :401–428, 1991.

[Bou04a] G. Boudol. ULM : A core programming model for global computing. In D. Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. Springer Berlin / Heidelberg, 2004.

[Bou04b] F. Boussinot. Reactive programming of cellular automata. 2004.

[Bou06] F. Boussinot. FairThreads : mixing cooperative and preemptive threads in C. *Concurrency and Computation : Practice and Experience*, 18(5) :445–469, 2006.

[Bou10] G. Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6) :716–736, 2010.

[BPR99] F. Bellifemine, A. Poggi, and G. Rimassa. JADE–a FIPA-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[Bré79] D. Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4) :251–256, 1979.

[BRS93] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 85–98, New York, NY, USA, 1993. ACM.

[BS89] P. Bogacki and L.F. Shampine. A 3(2) pair of runge - kutta formulas. *Applied Mathematics Letters*, 2(4) :321 – 325, 1989.

[BS95] E. Barendsen and S. Smetsers. Uniqueness type inference. In *PLILPS '95 : Proceedings of the 7th International Symposium on Programming Languages : Implementations, Logics and Programs*, pages 189–206, London, UK, 1995. Springer-Verlag.

[BS98] F. Boussinot and J.-F. Susini. The SugarCubes tool box : a reactive Java framework. *Software : Practice and Experience*, 28(14) :1531–1550, 1998.

[BSTH01] F. Boussinot, J.-F. Susini, F. Tran, and L. Hazard. A reactive behavior framework for dynamic virtual worlds. In *Virtual Reality Modeling Language Symposium : Proceedings of the sixth international conference on 3 D Web technology*, volume 2001, pages 69–75, 2001.

[CAC+81] G. J. Chaitin, M.A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1) :47 – 57, 1981.

[Cas92] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1) :125–140, 1992.

[CGP12] A. Cohen, L. Gérard, and M. Pouzet. Programming parallelism with futures in Lustre. In *ACM International Conference on Embedded Software (EMSOFT'12)*, Tampere, Finland, October 7-12 2012. ACM. Best paper award.

[CHT02] C. Calcagno, S. Helsen, and P. Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2) :199–221, 2002.

[CL05] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[Con63] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7) :396–408, 1963.

[Con08] A. Cont. ANTESCOFO : Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference (ICMC)*, 2008.

[CP96] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ICFP '96 : Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 226–238, New York, NY, USA, 1996. ACM.

[CP01] P. Cuoq and M. Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[CP03] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 134–155. Springer Berlin / Heidelberg, 2003.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : a declarative language for real-time programming. In *POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[CPP+02] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. Saxo-rt : Interpreting Esterel semantic on a sequential execution structure. *Electronic Notes in Theoretical Computer Science*, 65(5) :80 – 94, 2002. SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002).

[CPP05] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[Die89] P. F. Dietz. Fully persistent arrays (extended array). In *WADS '89 : Proceedings of the Workshop on Algorithms and Data Structures*, pages 67–74, London, UK, 1989. Springer-Verlag.

[DL93] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL)*, pages 113–123. ACM press, January 1993.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[DMR10] G. Delaval, H. Marchand, and É. Rutten. Contracts for modular discrete controller synthesis. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010)*, Stockholm, Sweden, April 2010.

[DPR01] A. Dovier, E. Pontelli, and G. Rossi. Set unification revisited, 2001.

[Edw00] S.A. Edwards. Compiling Esterel into sequential code. In *DAC '00 : Proceedings of the 37th Annual Design Automation Conference*, pages 322–327, New York, NY, USA, 2000. ACM.

[FG02] C. Fournet and G. Gonthier. The join calculus : A language for distributed mobile programming. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer Berlin / Heidelberg, 2002.

[FLFMS04] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml : A language for concurrent distributed and mobile programming. In J. Jeuring and S.P. Jones, editors, *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 1948–1948. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-44833-4_5.

[FW84] D.P. Friedman and M. Wand. Reification : Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 348–355. ACM, 1984.

[Gay08] S. Gay. Analyse d'échappement de portée en ReactiveML. Master's thesis, MPRI, 2008.

[GBS09] M. Gemünde, J. Brandt, and K. Schneider. Clock refinement in imperative synchronous languages. *SYNCHRON*, 9 :3–21, 2009.

[GBS10a] M. Gemunde, J. Brandt, and K. Schneider. Compilation of imperative synchronous programs with refined clocks. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 209–218. IEEE, 2010.

[GBS10b] M. Gemünde, J. Brandt, and K. Schneider. A formal semantics of clock refinement in imperative synchronous languages. In *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*, pages 157–168. IEEE, 2010.

[GBS11] M. Gemunde, J. Brandt, and K. Schneider. Causality analysis of synchronous programs with refined clocks. In *High Level Design Validation and Test Workshop (HLDVT), 2011 IEEE International*, pages 25–32. IEEE, 2011.

[GBS13] M. Gemunde, J. Brandt, and K. Schneider. Clock refinement in imperative synchronous languages. *EURASIP Journal on Embedded Systems*, 2013.

[Gel11] B. Gelineau. Programmation synchrone pour GPU. Master's thesis, Ecole Polytechnique, 2011.

[Gér08] L. Gérard. Des horloges entières pour la répartition de programmes synchrones flot de données. Master's thesis, Université Paris-Sud 11, 2008.

[GFW99] S.E. Ganz, D.P. Friedman, and M. Wand. Trampolined style. In *ACM SIGPLAN Notices*, volume 34, pages 18–27. ACM, 1999.

[GG10] A. Gamatié and T. Gautier. The Signal synchronous multiclock approach to the design of distributed embedded systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(5) :641–657, 2010.

[GGPP12] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages. In *LCTES'12*, 2012.

[GH88] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88 : Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, New York, NY, USA, 1988. ACM.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, PhD thesis, Universit e Paris VII, 1972.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1) :1–102, 1987.

[Gir05] A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science, New-York.

[GM13] A. Guatto and L. Mandel. Communication personnelle, 2013.

[GP10] A. Guatto and M. Pouzet. Rapport d'étude sur la traduction de SCADE/Lustre vers VHDL. Technical report, Laboratoire de Recherche en Informatique, 2010.

[GR97] J. Garrigue and D. Rémy. Extending ML with semi-explicit higher-order polymorphism. In *Theoretical Aspects of Computer Software*, pages 20–46. Springer, 1997.

[HB85] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *POPL '85 : Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 300–314, New York, NY, USA, 1985. ACM.

[HJ85] R.H. Halstead Jr. Multilisp : A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4) :501–538, 1985.

[Hoa78] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[HP85] D. Harel and A. Pnueli. On the Development of Reactive Systems. *Logics and models of concurrent systems*, page 477, 1985.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, Germany, August 1991.

[HS08] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[HSB99] L. Hazard, J.-F. Susini, and F. Boussinot. The Junior Reactive Kernel. Technical report, MEIJE - INRIA Sophia Antipolis - INRIA, 1999.

[Hue75] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1) :27–57, 1975.

[Jef13] A. Jeffrey. Functional reactive programming with liveness guarantees. In *ICFP'13*, 2013.

[JG91] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 303–310, New York, NY, USA, 1991. ACM.

[JGF96] S.P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Annual Symposium on Principles of Programming Languages : Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 21, pages 295–308. Citeseer, 1996.

[JVWS07] S.P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1) :1–82, 2007.

[LBR09] D. Le Botlan and D. Rémy. Recasting MLF. *Information and Computation*, 207(6) :726–785, 2009.

[Lee06] E.A. Lee. The problem with threads. *Computer*, 39(5) :33–42, 2006.

[Lei09] D. Leijen. Flexible types : robust type inference for first-class polymorphism. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 66–77. ACM, 2009.

[Ler09] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52 :107–115, July 2009.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[LGTLL03] P. Le Guernic, J.P. Talpin, and J.C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(03) :261–303, 2003.

[LO92] K. Laufer and M. Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, 1992.

[LP00] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2) :340–377, 2000.

[Man06] L. Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6 - Pierre et Marie Curie, 2006.

[MB05] L. Mandel and F. 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.

[MC05] J. Mikac and P. Caspi. Temporal refinement for Lustre. 2005.

[MGS10] A. Malik, A. Girault, and Z. Salcic. The DSystemJ programming language for dynamic GALS systems : its semantics, compilation, implementation, and run-time system. Research Report RR-7346, INRIA, 07 2010.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3) :348–375, 1978.

[MJT04] S. Marlow, S.P. Jones, and W. Thaller. Extending the Haskell foreign function interface with concurrency. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 22–32. ACM, 2004.

[MM09] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[Mor07] L. Morel. Array Iterators in Lustre : From a Language Extension to Its Exploitation in Validation. *EURASIP Journal on Embedded Systems*, 2007.

[MP94] M. Mauny and F. Pottier. An implementation of Caml-Light with existential types. 1994.

[MP05] L. Mandel and M. Pouzet. ReactiveML : a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

[MP08a] L. Mandel and F. Plateau. Interactive programming of reactive systems. In *Proceedings of Model-driven High-level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, April 2008.

[MP08b] L. Mandel and M. Pouzet. ReactiveML, un langage fonctionnel pour la programmation réactive. *Technique et science informatiques (TSI)*, 27(9-10) :1097–1128, 2008.

[MP13a] L. Mandel and C. Pasteur. Réactivité des systèmes coopératifs : le cas de ReactiveML. In *Vingt-quatrièmes Journées Francophones des Langages Applicatifs (JFLA 2013)*, Aussois, France, February 2013.

[MP13b] L. Mandel and C. Pasteur. Reactivity of cooperative systems : Application to ReactiveML. 2013.

[MPP13] L. Mandel, C. Pasteur, and M. Pouzet. Time refinement in a functional synchronous language. *PPDP'13*, 2013.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata : About modes and states for reactive systems. In *Programming Languages and Systems*, pages 185–199. Springer, 1998.

[MSRG10] A. Malik, Z. Salcic, P.S. Roop, and A. Girault. SystemJ : A GALS language for system level design. *Computer Languages, Systems & Structures*, 36(4) :317 – 344, 2010.

[NN93] F. Nielson and H. Nielson. From CML to process algebras. In *CONCUR'93*, pages 493–508. Springer, 1993.

[NN99] F. Nielson and H. Nielson. Type and effect systems. *Correct System Design*, pages 114–136, 1999.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer-Verlag New York Inc, 1999.

[Nov03] D. Novillo. TreeSSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[Ode91] M. Odersky. How to make destructive updates less destructive. In *POPL '91 : Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–36, New York, NY, USA, 1991. ACM.

[OL96] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67. ACM, 1996.

[Pag10] B. Pagano. The optimization of iterators and updates for functional arrays in SCADE 6. Personal communication, June 2010.

[Pan01] P.R. Panda. SystemC-a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pages 75–80. IEEE, 2001.

[Pas10] C. Pasteur. Optimisation des tableaux dans SCADE. Master's thesis, Ecole Polytechnique, 2010.

[Pie02] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[Pie05] B.C. Pierce. *Advanced topics in types and programming languages*. The MIT Press, 2005.

[Pla10] F. Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université de Paris-SUD 11, 2010.

[PT00] B.C. Pierce and D.N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1) :1–44, 2000.

[Rém94] D. Rémy. Programming objects with ml-art an extension to ml with abstract and record types. In *Theoretical Aspects of Computer Software*, pages 321–346. Springer, 1994.

[Rém05] D. Rémy. Simple, partial type-inference for System F based on type-containment. In *ACM SIGPLAN Notices*, volume 40, pages 130–143. ACM, 2005.

[Rep93] J. Reppy. Concurrent ML : Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, UK, 1974. Springer-Verlag.

[Rey87] C.W. Reynolds. Flocks, herds and schools : A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[RV97] D. Rémy and J. Vouillon. Objective ML : A simple object-oriented extension of ml. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53. ACM, 1997.

[SBS04] M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214. ACM, 2004.

[SBT96] T.R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference, 1996. ED&TC 96. Proceedings*, pages 328–333. IEEE, 1996.

[Sch09] K. Schneider. *The synchronous programming language Quartz*. Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[SGH93] P. Schnorf, M. Ganapathi, and J. L. Hennessy. Compile-time copy elimination. *Softw. Pract. Exper.*, 23(11) :1175–1200, 1993.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1) :51–92, 1993.

[Sij89] B.A. Sijtsma. On the productivity of recursive list definitions. *TOPLAS'89*, 11(4) :633–649, 1989.

[SMMM06] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. GLONEMO : global and accurate formal models for the analysis of ad-hoc sensor networks. In *InterSense '06 : Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 3, New York, NY, USA, 2006. ACM.

[SPL11] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. *Practical Aspects of Declarative Languages*, pages 175–189, 2011.

[SSVH08] C. Skalka, S. Smith, and D. Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(02) :179–249, 2008.

[Sus01] J.-F. Susini. *L'Approche Réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior*. PhD thesis, Ecole des Mines de Paris, 2001.

[TB98] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20 :724–767, July 1998.

[TdS05] O. Tardieu and R. de Simone. Loops in Esterel. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4) :708–750, 2005.

[Thi13] V. Thiberville. Coûts et retards associés au sur-échantillonnage dans un langage synchrone. Master's thesis, Ecole Polytechnique, 2013.

[TJ92] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162 –173, jun 1992.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and computation*, 89(1) :1–34, 1990.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2) :109–176, 1997.

[Vou08] J. Vouillon. Lwt : a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12. ACM, 2008.

[VWPJ08] D. Vytiniotis, S. Weirich, and S. Peyton Jones. FPH : First-class polymorphism for Haskell. In *ACM Sigplan Notices*, volume 43, pages 295–306. ACM, 2008.

[Wad88] P. Wadler. Deforestation : transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, 1988. North-Holland Publishing Co.

[Wad90] P. Wadler. Linear types can change the world ! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[Wan87] M. Wand. Complete type inference for simple objects. In *LICS*, volume 87, pages 37–44, 1987.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3) :36, 2008.

[Wel99] J.B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1) :111–156, 1999.

Index

Valeurs

Piles

Sémantique comportementale

Sémantique opérationnelle

Calcul d'horloges

Comportements

Analyse de réactivité

Cédric PASTEUR

Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel

Résumé

Nous nous intéressons dans ce manuscrit au langage REACTIVEML, qui est une extension de ML avec des constructions inspirées des langages synchrones. L'idée de ces langages est de diviser l'exécution d'un programme en une suite d'instants logiques discrets. Cela permet de proposer un modèle de concurrence déterministe que l'on peut compiler vers du code séquentiel impératif. La principale application de REACTIVEML est la simulation discrète, par exemple de réseaux de capteurs. Nous cherchons ici à programmer des simulations à grande échelle, ce qui pose deux questions : sait-on les programmer de façon simple et modulaire ? sait-on ensuite exécuter ces programmes efficacement ?

Nous répondons à la première question en proposant une extension du modèle synchrone appelée *domaines réactifs*. Elle permet de créer des instants locaux invisibles de l'extérieur. Cela rend possible le raffinement temporel, c'est-à-dire le remplacement d'une approximation d'un système par une version plus détaillée sans changer son comportement externe. Nous développons dans ce manuscrit la sémantique formelle de cette construction ainsi que plusieurs analyses statiques, sous forme de systèmes de types-et-effets, garantissant la sûreté des programmes dans le langage étendu.

Enfin, nous montrons également plusieurs implémentations parallèles du langage pour tenter de répondre à la question du passage à l'échelle des simulations. Nous décrivons tout d'abord une implémentation avec threads communiquant par mémoire partagée et basée sur le vol de tâches, puis une seconde implémentation utilisant des processus lourds communiquant par envoi de messages.

Mots-clés : Concurrence ; Raffinement ; Langages synchrones ; Langages fonctionnels ; Sémantique ; Systèmes de types-et-effets ; Parallélisme

Abstract

In this thesis, we are interested in the REACTIVEML language, which extends ML with constructs inspired from synchronous languages. The idea of these languages is to divide the execution of a program into a succession of discrete logical instants. It results in a deterministic model of concurrency that can be compiled to sequential imperative code. The main application domain of REACTIVEML is discrete simulation, for instance of sensor networks. We focus here on the problem of large scale simulation that raises two questions: can we program such simulations easily and modularly? can we execute them efficiently?

We answer the first question by proposing an extension of the synchronous model called *reactive domains*. It allows the creation of local instants, invisible from the outside. It enables temporal refinement, that is, to replace an approximate model of a system with a more detailed version without changing its external behavior. We describe the formal semantics of this construct as well as static analyses, presented as type-and-effect systems, to ensure the soundness of programs in the extended setting.

Our second contribution is several parallel implementations of the language to allow the scaling of simulations. We first describe an implementation based on threads communicating by shared memory and using work stealing, and then a second one based on processes communicating by message passing.

Keywords: Concurrency; Refinement; Synchronous languages; Functional languages; Semantics; Type-and-effect systems; Parallelism