Programmation synchrone

Adrien Guatto

2020–2022

Ces notes de cours proposent une introduction à la programmation des systèmes réactifs par le biais des langages de programmation dits *synchrone*. Elles correspondent à un enseignement délivré durant les années universitaires 2020–2022 à l'Université de Paris, en Master 2 Informatique ainsi qu'à l'École d'Ingénieur Denis Diderot.

Version du 14 octobre 2022. Je suis preneur de toute coquille, erreur ou remarque par courriel à l'adresse adrien@guatto.org.

Table des matières

Introduction

L'apprentissage de la programmation se structure traditionnellement le long de deux axes :

1. la pratique de la programmation dans des langages variés, par exemple Java, C ou OCaml ;
2. l'étude de concepts et techniques algorithmiques indépendants du langage de programmation, par exemple l'algorithmique des tris, ou encore celle des graphes.

Le but du présent cours est de prolonger ces deux axes dans une direction qui devrait être nouvelle pour vous : celle des programmes *réactifs*, par opposition aux programmes *transformationnels* [1].

1. La distinction entre ces deux classes de programmes est traditionnellement attribuée à Zohar Manna et Amir Pnueli [11].

Programmes et systèmes réactifs

Un programme transformationnel lit une entrée, la traite, puis produit un résultat complet en temps fini. L'utilitaire `sort` d'UNIX, qui trie une liste de lignes par ordre alphabétique, constitue un exemple très simple de programme transformationnel. Un exemple plus sophistiqué est fourni par n'importe quel compilateur capable de traduire un fichier source en un fichier en langage cible [2] réutilisable [3]. La plupart des programmes que vous avez écrits jusqu'ici sont transformationnels.

Par opposition, un **programme réactif est en interaction continuelle avec un environnement extérieur** qu'il va chercher à contrôler, surveiller ou réguler. Cet environnement extérieur est généralement un environnement physique, que notre programme observe par le biais de capteurs et influence par le biais d'actuateurs — voir figure 1. Pensez au pilote automatique d'un avion (*fly-by-wire* en anglais), au *firmware* du modem 4G de votre téléphone, ou plus modestement au contrôleur de votre four à micro-ondes.

2. Par exemple, le langage machine compris par votre processeur pour `gcc`, ou le code-octet de la machine virtuelle Java pour `javac`.
3. Le cas des compilateurs à la volée (*just-in-time*), qu'on trouve notamment dans les navigateurs web, est plus complexe et ne rentre pas facilement dans la dichotomie transformationnel *vs.* réactif.

Question 1. *Pouvez-vous citer d'autres exemples de programmes transformationnels ? De programmes réactifs ? De programmes qui ne semblent pas appartenir nettement à un des deux cotés de cette classification ?*

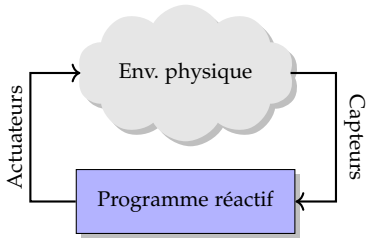


FIGURE 1: système réactif générique.

Un programme réactif ne peut généralement pas être considéré en isolation de son environnement extérieur. Par exemple, le pilote automatique d'un avion fait des hypothèses sur l'environnement aérien (altitude, force du vent, etc.) ainsi que sur l'avion lui même (poids, portance, etc.). Pour cette raison, on parlera généralement de *système réactif* pour englober sous un même terme le logiciel et son environnement, en insistant sur leur interdépendance [4].

En plus de cette définition générale, beaucoup de systèmes réactifs partagent un petit nombre de caractéristiques essentielles, que nous allons maintenant aborder.

4. On verra ultérieurement que cette interdépendance se traduit de façon concrète par la pratique qui consiste à développer simultanément le programme réactif et un modèle logiciel de son environnement physique, afin de simuler leur interaction.

Criticité. Certains programmes réactifs contrôlent des dispositifs physiques où l'erreur peut avoir des conséquences catastrophiques sur la vie humaine, ou bien un coût financier démesuré. Citons deux erreurs restées tristement célèbres.

— Le 4 juin 1996, le premier vol d'Ariane 5 aboutit à la destruction du lanceur 37 secondes après son décollage (figure 2). Une partie du logiciel avait été reprise d'Ariane 4 sans être mise à jour pour tenir compte de l'évolution des caractéristiques physiques du lanceur [5] .

— Entre 1985 et 1987, les machines de radiothérapie Therac-25 (figure 3) ont causé six irradiations massives, dont trois mortelles. Le tout a été causé par la combinaison d'une pratique défaillante du génie logiciel (manque de test), de l'absence de protections physiques (jugées redondantes), et de la présence de bogues de programmation concurrente de type "condition de course" (*race condition*).

Ces systèmes réactifs, où l'erreur est inadmissible, sont dits *critiques*. Leur conception et réalisation doit assurer un haut niveau de sûreté, et exige donc l'emploi d'une méthodologie adaptée.

FIGURE 2: vol 501 d'Ariane 5.

5. On peut approfondir sur les causes logicielles de cet échec et leurs conséquences matérielles en lisant le rapport de la commission d'enquête. Une copie est disponible en ligne.

Question 2. *Pouvez-vous citer un exemple de système critique dans le secteur des transport ? Dans le secteur bio-médical ? Dans le secteur militaire ?*

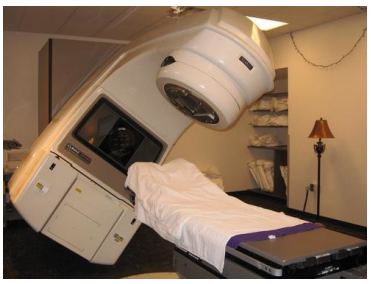


FIGURE 3: machine Therac-25.

Contraintes temporelles. Les programmes réactifs sont souvent soumis à des contraintes dites de *temps-réel* — c'est presque toujours le cas s'ils appartiennent à un système critique. Un programme temps-réel doit absolument réagir en un temps borné maximal à certains changements de l'environnement. Manquer cette échéance peut entraîner un échec catastrophique de tout le système. Par exemple, le système TCAS II, employé dans l'aviation civile, est chargé d'avertir un pilote en cas de présence d'un appareil intrus dans une de ses trois zones d'intérêt (cf. figure 4). Le pilote doit être averti très rapidement pour lui laisser le temps de réagir avant qu'une collision ait pu se produire.

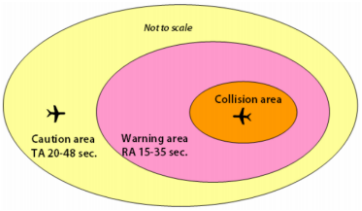


FIGURE 4: Zones d'intérêt du système d'évitement de collision TCAS II.

Un des facteurs principaux qui impose des contraintes temps-réel aux programmes réactifs est leur interaction avec l'environnement physique. L'évolution de celui-ci est régie par des lois mathématiques qui évoluent en *temps continu*. À l'inverse, l'exécution du programme réactif prend la forme d'une séquence de transitions discrètes, autrement dit, en *temps discret*. Dès lors, il faut s'interroger sur la capacité du programme réactif à se faire une idée juste du système physique. Il s'agit d'un problème *d'échantillonage* : le programme réactif doit observer le système physique à la bonne fréquence, ni trop lente, ni trop rapide. Autrement dit, la fréquence de réaction du programme fait partie intégrante de l'algorithme employé [6].

6. La littérature sur les systèmes temps-réel distingue souvent la *correction fonctionnelle* des *contraintes temps-réel*, et propose de les considérer indépendamment. Cet exemple montre que cette approche n'est pas toujours pertinente.

Mathématiques dédiées. Le paragraphe précédent illustre l'importance des mathématiques appliquées dans la conception d'un programme réactif en interaction avec un environnement. En général, cet environnement évolue au cours du temps selon ses propre lois — on parle alors de *système dynamique*. La sous-discipline des mathématique appliquées qui se consacre au contrôle des systèmes dynamiques s'appelle l'*automatique* (*control theory* en anglais). Ce contrôle peut passer par des dispositifs numériques, mais aussi purement électriques ou mécaniques [7]. Les théorèmes d'automatique fournissent par exemple des garanties sur la correction de l'échantillonage, comme nous en avons discuté ci-dessus, mais aussi des outils d'analyse du comportement des systèmes dynamiques.

Le présent cours n'est pas un cours de mathématiques, aussi nous ne traiterons pas d'automatique à proprement parler. Toutefois, nous ferons référence à certains de ses résultats à plusieurs reprises, et nous montrerons comment certains concepts issus de l'automatique sont utiles pour la programmation de systèmes réactifs, sans détailler leurs sous-bassements théoriques [8].

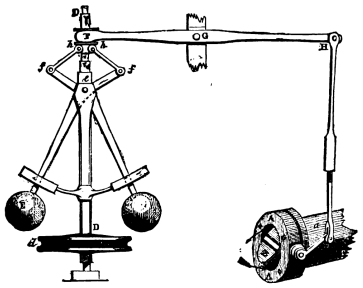
Contraintes de ressources. Enfin, les programmes réactifs se retrouvent en général exécutés par du matériel informatique dont c'est la seule fonction. On parle typiquement de *système embarqué*. De tels systèmes sont généralement soumis à des impératifs de coûts forts, surtout lorsqu'ils doivent être produits à de nombreux exemplaires dans une perspective commerciale (pensons, par exemple, aux microcontrôleurs qu'on trouve dans les fours à micro-ondes actuels). Ces impératifs de coût exigent généralement des programmes réactifs qu'ils soient économes en temps, en espace, en énergie.

La programmation des systèmes embarqués, envisagée sous l'angle de l'optimisation de l'usage des ressources, est un sujet riche, connecté à de nombreux sous-domaines de l'informatique dont la compilation, les systèmes d'exploitation ou l'architecture des processeurs. Dans ce cours, nous nous focaliserons néanmoins sur les aspects de haut niveau (expressivité, sûreté) de la programmation réactive dans la mesure où ses aspects de bas niveau sont traités dans d'autres cours.

Langages synchrones

On a vu que les programmes réactifs peuvent être critiques, doivent généralement obéir à des contraintes temporelles et de ressources, et reposent sur des théories mathématiques dédiées, notamment l'automatique. Il est donc légitime pour un langage de programmation dédié aux systèmes réactifs de chercher à faciliter l'écriture de programmes mettant en jeu des procédés de contrôle issus de l'automatique, tout en

7. Un exemple célèbre de dispositif de contrôle mécanique est le régulateur à boules de James Watt.



Il permet de maintenir une machine à vapeur à une vitesse quasiment constante en contrôlant l'arrivée de vapeur. Celle-ci diminue si la machine est trop rapide, et augmente si elle est trop lente.

8. Si vous souhaitez découvrir ceux-ci, le livre d'Åström et Murray [12] offre une introduction à l'automatique illustrée de nombreux exemples. La deuxième édition est disponible en ligne.

assurant un niveau de sûreté élevé, et en étant économe en ressources.

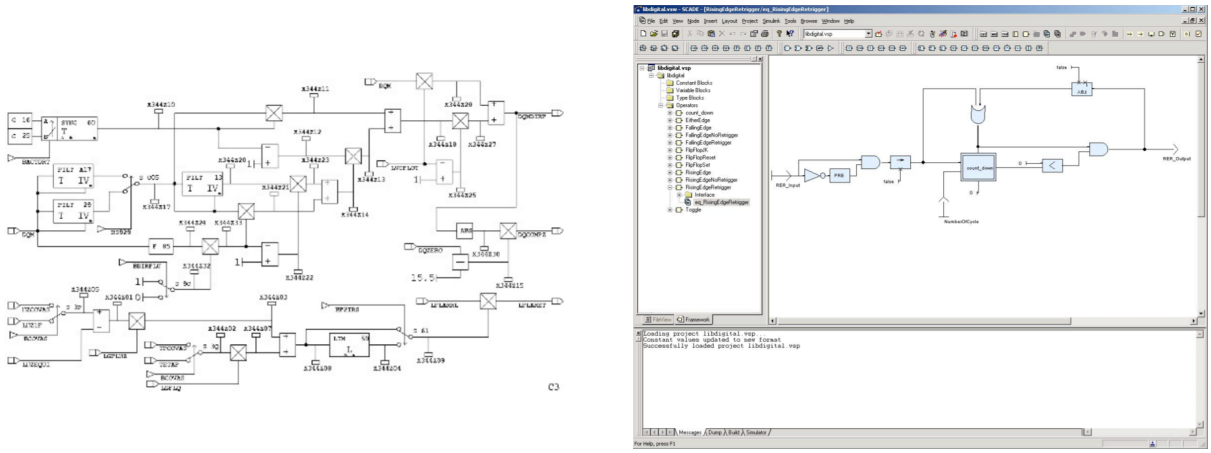


FIGURE 5: les langages synchrones à flots de données, du protolangage Spécification Assistée par Ordinateur (SAO) d'Airbus (à gauche), au langage industriel contemporain SCADE 6, développé par la compagnie Ansys (à droite).

Les langages de programmation synchrones suivent une telle voie. Issus de la recherche en informatique française, allemande et américaine depuis les années 1980, ils reposent sur une notion de temps discret global qui facilite la mise au point des programmes réactifs. Leur usage est désormais routinier, notamment dans l'industrie du transport. À titre d'exemple, le langage synchrone SCADE 6 a servi à la réalisation des commandes de vol du célèbre Airbus A380.

Si plusieurs familles de langages synchrones existent, nous nous focaliserons sur celle des langages synchrones à *flots de données* (*data-flow* en anglais) [9]. Ceux-ci sont en effet largement les plus utilisés dans l'industrie, tout en restant en développement actif dans le monde universitaire. Ils facilitent la programmation réactive en offrant des concepts proches à la fois de l'automatique et de la programmation fonctionnelle. Ils sont issus de travaux pionniers de chercheurs grenoblois, qui ont voulu comprendre et généraliser la méthodologie empirique suivie par les ingénieurs pour concevoir les premiers dispositifs de contrôle numériques [10]. En une quarantaine d'années, ces langages ont contribué à l'évolution graduelle d'une méthodologie basée sur des schémas informels, suggestifs mais sans contenu calculatoire, à de véritables langages de programmation rigoureux, dotés d'une sémantique solide et de compilateurs sophistiqués (figure 5).

9. On pourra trouver un panorama historique des autres familles de langages synchrones dans l'article de Benveniste et al. [1].

10. On peut lire l'article [3] original de ces chercheurs au sujet du langage Lustre, l'ancêtre commun de tous les langages synchrones à flots de données.

Le reste du cours

Le but de ce cours est donc d'offrir une introduction aux systèmes réactifs ainsi qu'aux langages synchrones. Il sera structuré selon les deux axes décrits au début de ce texte, programmation et algorithmique, auxquels on adjoindra un troisième sujet, l'implémentation des langages synchrones, qui occupera la fin du semestre.

Tout d'abord, on pratiquera la programmation dans un langage synchrone à flots, le langage Heptagon. Heptagon est un langage très proche de SCADE 6, mais développé par des universitaires. Il dispose de fonctionnalités modernes : compilation séparée, automates imbriqués, gestion des tableaux. Sa proximité vis-à-vis de SCADE vous assure que vos compétences seront transférables facilement. De plus, il s'agit d'un logiciel libre d'une installation simple.

Vous pouvez déjà avoir un aperçu du langage en consultant son site.

http://heptagon.gforge.inria.fr

Dans un second temps, nous discuterons des bases d'automatique appliquée qui servent de soubassements algorithmiques à la plupart des programmes réactifs. Nous n'entrerons quasiment pas dans les détails mathématiques, adoptant à la place une démarche expérimentale basée sur la programmation de petits contrôleurs en Heptagon.

Enfin, nous terminerons le semestre avec une introduction à l'implémentation des langages synchrones. Il s'agit d'étudier les analyses statiques et techniques de génération de code employées par les compilateurs, les secondes reposant sur les premières. Pour bien comprendre leur fonctionnement, une connaissance minimale de la sémantique formelle des langages synchrones est indispensable, et celle-ci sera donc introduite.

La programmation synchrone à flots de données

On a vu qu'un programme réactif est en interaction continue avec un environnement physique via capteurs et actuateurs, comme illustré de façon schématique par la figure 1. Les langages synchrones partent du principe que l'exécution d'un tel programme se produit de façon *cyclique*, c'est à dire comme une suite d'interactions entre le programme et son environnement. Un programme réactif n'agit en général pas uniquement en fonction de la valeur courante des capteurs, mais également de celles recueillies durant les interactions précédentes : il doit donc, pour ce faire, avoir accès à une *mémoire* dont le contenu persiste d'une interaction sur l'autre. De façon générale, chaque interaction se divise en trois phases distinctes, *lecture-calcul-écriture* :

L. lecture des capteurs et de l'état courant de la mémoire ;

C. phase de calcul des consignes des actuateurs et du nouvel état ;

E. positionnement des actuateurs et mise à jour de la mémoire.

Enfin, les langages synchrones ne se préoccupent pas des détails bas-niveau d'accès aux capteurs et actuateurs. Il est donc utile de s'abstraire de la nature physique de l'environnement — sans toutefois oublier qu'elle induit les contraintes temporelles discutées précédemment. Une fois adopté ce point de vue simplifié, la structure d'un système réactif cyclique peut être représentée comme à la figure 6, où capteurs et actuateurs ont été remplacés par des entrées et sorties génériques.

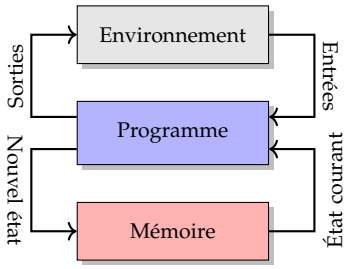


FIGURE 6: système réactif cyclique.

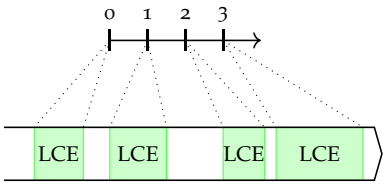


FIGURE 7: temps logique, temps physique.

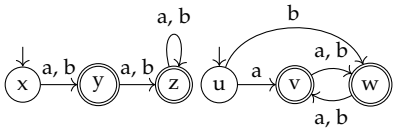
Le choix de structurer l'exécution comme une suite infinie d'interactions introduit naturellement une notion de *temps logique*. Ce temps logique est constituée d'une succession d'*instants logiques*, chacun d'eux correspondant à une interaction entre programme et environnement. Le caractère *logique* de cette temporalité réside dans l'omission volontaire du temps d'exécution. En d'autres termes, **un instant logique n'a pas d'épaisseur : du point de vue synchrone, l'interaction avec l'environnement est instantanée**. On appelle ce principe l'*hypothèse synchrone*. La figure 7 illustre la relation entre temps logique et temps physique : le cycle Lecture-Calcul-Écriture (LCE) réalisé à chaque interaction peut prendre un temps d'exécution variable. De plus, rien ne garantit que les interactions soient exécutées à intervalle régulier.

On peut trouver l'hypothèse synchrone surprenante dans la mesure où, comme on l'a vu, de nombreux systèmes réactifs sont soumis à des contraintes temporelles strictes. On verra que ce choix simplifie en réalité leur mise au point en repoussant la prise en compte du temps physique aussi longtemps que possible durant le développement.

Les langages synchrones font donc le choix d'une exécution cyclique et d'une structuration du temps comme succession d'instants logiques. À un certain niveau d'abstraction, un programme synchrone peut donc être vu comme une machine à état, dont chaque transition correspondrait à un instant logique. Comme on cherche un formalisme mathématiquement simple pour décrire ces programmes, il est naturel de se tourner vers la théorie des automates et, plus précisément, des *transducteurs*, c'est à dires des automates finis qui, en plus de recevoir un mot en entrée, produit également un mot en sortie. Toutefois, les automates restent des objets complexes : par exemple, raisonner sur l'équivalence entre automates passe naturellement par la *bisimulation* (cf. figure 8), une notion profonde mais relativement technique. Une autre difficulté est celle de la modularité : si on peut définir diverses manières d'assembler plusieurs automates en un automate plus gros — par exemple par la composition séquentielle de transducteurs — le résultat sera un automate à plat, sans structure. En bref, si le formalisme des automates est indispensable à la vérification de systèmes réactifs cycliques, il ne semble pas fournir un *langage* adapté au génie logiciel, du moins s'il n'est pas complété par d'autres principes de structuration.

On peut contraster la notion d'automate à une autre notion mathématique qui, bien qu'élémentaire, s'est révélée très compatible avec le génie logiciel : celle de fonction. Une fonction est un objet plus simple qu'un automate au sens où l'égalité entre fonctions est triviale — par définition, deux fonctions sont égales lorsqu'elles envoient les mêmes entrées vers les mêmes sorties. De plus, les langages dits "fonctionnels" tels que Haskell ou OCaml ont montré comment bâtir un langage de programmation au dessus de la notion de fonction. Il est donc naturel

Les deux automates ci-dessous sont-ils équivalents, c'est à dire, reconnaissent-ils le même langage ? (Cet exemple simple est issu d'un article de Bonchi et Pous [2].)



On peut étudier cette question à l'aide de la notion de bisimulation. Écrivons S pour l'ensemble des états de nos automates. Une bisimulation est une relation $R \subseteq S \times S$ telle que, pour toute paire d'états $(s_1, s_2) \in R$, on ait :

1. s_1 est final ssi s_2 est final,
2. si $s_1 \xrightarrow{a} s_1'$ alors il existe s_2' tel que $s_2 \xrightarrow{a} s_2'$ et $(s_1', s_2') \in R$,
3. si $s_2 \xrightarrow{a} s_2'$ alors il existe s_1' tel que $s_1 \xrightarrow{a} s_1'$ et $(s_1', s_2') \in R$.

On peut montrer que deux états s_1, s_2 d'automates finis *déterministes* reconnaissent le même langage ssi il existe une bisimulation R telle que $(s_1, s_2) \in R$. Dans le cas qui nous occupe, existe-t-il une bisimulation R telle que $(x, u) \in R$? Oui ! Essayez de définir $R = \{(x, u), \dots\}$ en énumérant les couples manquants.

FIGURE 8: bisimulations entre automates.

de chercher à concilier le monde des automates avec celui des fonctions.

Pour ce faire, on peut partir d'une observation simple. Si Σ_1 est l'alphabet du mot d'entrée et Σ_2 l'alphabet du mot de sortie, un transducteur déterministe [11] A implémente une fonction $f_A : \Sigma_1^* \to \Sigma_2^*$ telle que $f_A(w) = w'$ lorsque A produit le mot w' en lisant le mot w. Cette fonction a toujours plusieurs propriétés remarquables, notamment son caractère *synchrone* : elle associe toujours un mot de longueur n à un mot de longeur n. L'idée clef des langages synchrones dits *à flots de données* est de partir de la fonction synchrone pour aller vers l'automate, plutôt que l'inverse. Autrement dit, un programme va consister en une fonction synchrone, et c'est le compilateur qui va reconstruire le transducteur sous-jacent, celui-ci étant vu comme une implémentation concrète de la fonction (cf figure 9). De plus, comme on s'intéresse aux systèmes réactifs, qui s'exécutent sans discontinuer, la fonction synchrone va consommer et produire non pas des mots finis dans Σ_1^* et Σ_2^*, mais des suites infinies de lettres, aussi appelées *flots*, et dont les ensembles sont dénotés Σ_1^ω et Σ_2^ω. On espère ainsi bénéficier du meilleur des deux mondes : la proximité des automates avec le modèle d'exécution sous-jacent, et l'expressivité des langages fonctionnels [12].

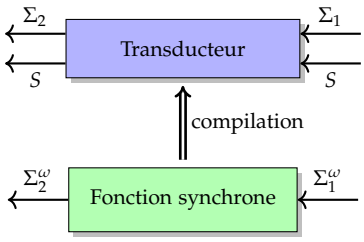


FIGURE 9: compilation des langages synchrones à flots de données.

11. Cette fonction peut être partielle si le transducteur est incomplet. Plus généralement, un transducteur non-déterministe donne lieu à une relation $R_A \subseteq \Sigma_1^* \times \Sigma_2^*$. Les relations implémentables par des transducteurs sont dites *rationnelles* (ou *régulières*), par analogie avec les langages et expressions rationnelles (ou régulières).

12. L'idée qui consiste à décrire des systèmes réactifs par des fonctions de flots est issue du travail pionnier de Kahn [9].

Premiers programmes

Nous allons maintenant explorer les langages synchrones à flots de données de façon concrète. Notre véhicule pour ce faire sera le langage Heptagon, qui est très proche du langage industriel SCADE 6 mais dont le compilateur est un logiciel libre.

Nœuds. Un programme Heptagon est un ensemble de fonctions synchrones sur les flots de données. On appelle ces fonctions des *nœuds*. Chaque nœud dispose d'une *interface*, liste finie de sorties et d'entrées déclarées avec leurs types, ainsi que d'un *corps*, qui est une liste d'équations définissant la valeur des sorties en fonction de celles des entrées. On peut par exemple définir un nœud correspondant à la fonction identité sur les entiers comme ci-dessous.

```
node identite(x : int) returns (y : int)
let
  y = x;
tel
```

Il est important de remarquer que le type `int`, en Heptagon, ne désigne pas un unique entier, mais un *flot* d'entiers. Il en va de même pour les types `float`, `bool`, etc. Le nœud ci-dessous représente donc la fonction mathématique $id : \mathbb{Z}_{32}^\omega \to \mathbb{Z}_{32}^\omega$, où $\mathbb{Z}_{32}$ désigne l'ensemble des entiers relatifs représentable en complément à deux sur 32 bits [13].

On peut représenter le comportement d'un nœud sur une entrée choisie à l'aide d'un *chronogramme*, c'est à dire d'un tableau dont chaque colonne correspond à un instant logique distinct.

x	-2	0	1	-3	2	2	...
y	-2	0	1	-3	2	2	...

Ce chronogramme illustre le caractère synchrone de la fonction identité, vue comme agissant sur des flots : les n premières valeurs de y dépendent uniquement des n premières valeurs de x. On verra que ce sera aussi le cas de fonctions bien plus complexes.

13. En réalité, Heptagon ne fixe pas la taille des entiers utilisés, mais aligne son type `int` au type `int` du langage C. Sa taille dépend donc de votre machine, et plus précisément du compilateur C utilisé pour compiler le code généré par Heptagon (cf. plus bas).

On peut compiler un programme Heptagon en demandant au compilateur `heptc` de produire une sortie en langage C. Les fichiers sources ainsi générés contiennent une implémentation du transducteur sous une forme ressemblant [14] au code ci-dessous.

14. En pratique, le compilateur applique certaines transformations qui rendent le code moins lisible mais plus efficace et plus court.

```
/* Définition de l'état du transducteur. */
struct identite_state { };
/* Défininition de la fonction d'initialisation de l'état. */
void identite_reset(struct identite_state *state) { }
/* Défininition de la fonction de transition. */
void identite_step(struct identite_state *state,
                   int x, int *y) { *y = x; }
```

On verra lors des cours et séances de travaux pratiques suivants une façon commode d'exécuter la fonction de transition.

Un nœud Heptagon peut également disposer de variables *locales*, qui ne sont ni des entrées ni des sorties. Elles doivent être déclarées avec le mot clef **var** et définies dans le corps du nœud.

```
node identite_bis(x : int) returns (y : int)
var z : int;
let
  z = x;
  y = z;
tel
```

Un point très important, commun à tous les langages synchrones à flots de données, est que l'ordre des définitions n'importe pas. Ainsi, on peut réécrire notre fonction identité de façon strictement équivalente mais en définissant `y` avant `z`.

```
node identite_ter(x : int) returns (y : int)
var z : int;
let
  y = z;
  z = x;
tel
```

Cet exemple montre que le point-virgule qui sépare les équations n'a rien à voir avec une construction de séquentialisation, comme en C, Java ou OCaml. Au contraire, il s'agit simplement de marquer la fin d'une équation dans le bloc de définitions *mutuellement récursives* compris entre **let** et **tel**.

La possibilité d'écrire des équations mutuellement récursives est nécessaire pour pouvoir écrire des fonctions de flots générales, comme on le verra ultérieurement. Elle ouvre néanmoins la porte à la possibilité d'erreurs. Considérons le code ci-dessous, en apparence une simple modification du précédent.

```
node identite_bad(x : int) returns (y : int)
var z : int;
let
  y = z;
  z = y;
tel
```

Ce programme est très suspect : on a défini `y` en fonction de `z`, et vice-versa ! Si l'on essaie de le compiler avec `heptc`, on obtient un message d'erreur.

```
$ heptc -target c ex-04-bad.ept
Causality error: the following constraint is not causal.
^z < y || ^y < z
```

On appelle les erreurs causées par ce genre de définitions circulaires, ou *cercles vicieux*, des erreurs de *causalité* [15]. L'étude de la notion de causalité est au coeur des langages synchrones, et il faut en comprendre le fonctionnement général pour programmer productivement dans un langage comme Heptagon. On y reviendra en détail lors des cours suivants, y compris une explication de ce message d'erreur.

15. On trouve parfois employé le terme plus sobre de *productivité*.

Programmation flots de données et causalité

Opérations combinatoires. On a vu que tout programme Heptagon manipule des *flots* de données, c'est à dire des suites infinies de valeurs. Tout comme les types `int` ou `bool` désignent respectivement les flots d'entiers et de booléens, en Heptagon les littéraux désignent des flots constants.

```
node f() returns (x, y : int; z : bool)
let
  x = 1;
  y = 42;
  z = false;
tel
```

Un nœud Heptagon peut avoir plusieurs entrées et plusieurs sorties. On peut grouper les déclarations successives des varibales de même type en séparant les noms de variables par des virgules, et les groupes de variables de même type par des points-virgules. Ainsi, `x, y : int; z : int` est équivalent à `x : int; y : int; z : int`.

Ainsi, dans le nœud ci-dessus, les littéraux `0`, `42` ou `false` désignent des flots constants. Les trois sorties `x`,`y` et `z` sont donc décrites par le chronogramme ci-dessous.

x	1	1	1	1	1	1	1	1	1	...
y	42	42	42	42	42	42	42	42	42	...
z	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	...

Pour formuler précisément les constructions d'un exemple, il est utile d'employer une notation formelle pour désigner le flot associé à une expression Heptagon. Autrement dit, on va distinguer la *sémantique*

d'une expression de sa *syntaxe*. Si e est une expression Heptagon, on écrira $[\![e]\!]$ pour l'objet sémantique associé. Il s'agira généralement d'un flot ou d'un n-uplet de flots. La sémantique $([\![l]\!]_n)_{n\in\mathbb{N}}$ d'un littéral l est un flot dont le $n$ème élément est défini par l'équation

$$[\![l]\!]_n = l.$$

Un nœud Heptagon est une fonction de flots, et peut donc être appliqué à des arguments pour produire des résultats. Ainsi, on peut appeler le nœud précédent depuis un autre nœud situé plus bas dans le même fichier [16].

```
node g() returns (o : int)
var x, y : int; z : bool;
let
  (x, y, z) = f();
  o = x + y;
tel
```

Le nœud `g`, en plus de sa sortie `o`, déclare trois variables locales `x,y,z` qui servent à stocker les résultats de `f`. La variable `z` n'est pas utilisée. La sortie de `g` est définie comme la somme des deux premières sorties de `f`. En Heptagon, la somme agit point à point sur les flots, tout comme les autres opérateurs binaires — soustraction, multiplication, division, opérateurs logiques et de comparaison, etc. On peut formaliser ce comportement par les équations sémantiques ci-dessous. La dernière d'entre-elles, écrite pour une opération binaire op quelconque, généralise les autres.

$$\begin{aligned}
[\![e_1 + e_2]\!]_n &= [\![e_1]\!]_n + [\![e_2]\!]_n \\
[\![e_1 - e_2]\!]_n &= [\![e_1]\!]_n - [\![e_2]\!]_n \\
&\dots \\
[\![op(e_1, e_2)]\!]_n &= op([\![e_1]\!]_n, [\![e_2]\!]_n)
\end{aligned}$$

En appliquant ces définitions au nœud `g`, on obtient

$$\begin{aligned}
[\![\mathtt{o}]\!]_n &= [\![\mathtt{x + y}]\!]_n \\
&= [\![\mathtt{x}]\!]_n + [\![\mathtt{y}]\!]_n \\
&= 1 + 42 \\
&= 43.
\end{aligned}$$

La sortie de `g` est donc le flot constant 43.

Tout comme les opérateurs arithmétiques et logiques, la construction **`if/then/else`** d'Heptagon fonctionne de façon point à point.

$$[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!]_n = \begin{cases} [\![e_2]\!]_n \text{ si } [\![e_1]\!]_n = \mathit{true} \\ [\![e_3]\!]_n \text{ si } [\![e_1]\!]_n = \mathit{false} \end{cases}$$

16. En Heptagon, chaque fichier donne lieu à un *module* distinct. On peut faire référence à un nœud situé dans un autre module en le préfixant par le nom du module en question. Par exemple, si le nœud `f` a été défini dans le fichier `a.ept`, on peut y faire référence depuis un fichier `b.ept` en écrivant `A.f`. Le fichier `a.ept` doit avoir été compilé avant `b.ept` de sorte à produire un fichier d'interface `a.epci`. Si `a.epci` est présent dans un répertoire `DIR` autre que `b.ept`, on peut indiquer ce chemin via `heptc -I DIR`.

Question 3. *Supposons qu'on fournisse comme entrée* `k` *au nœud défini ci-dessous le flot constant* `12`*. Que vaut la sortie* `o` *?*

```
node h(k : int) returns (o : int)
var x, y : int; z : bool;
let
  (x, y, z) = f();
  o = k + if z then 2 * x else y;
tel
```

Opérateurs séquentiels. Jusqu'ici, nous n'avons écrits que des nœuds manipulant des flots constants, et des opérateurs dont la valeur du flot de sortie à l'instant courant ne dépend que des valeurs des flots d'entrée à l'instant courant. De tels opérateurs sont dits *combinatoires*. Ce n'est pas très excitant : on a essentiellement écrit des expressions arithmétiques et booléennes où le temps ne joue aucun rôle. La première construction avec une comportement temporel non-trivial que nous allons étudier sera l'opérateur binaire **`fby`**. Un tel opérateur est dite *séquentiel*. Sa sémantique est donnée par les équations suivantes.

Pour demander au compilateur Heptagon de vérifier qu'un nœud est combinatoire, on peut le définir à l'aide du mot-clef **`fun`** plutôt que **`node`**.

$$[\![e_1 \texttt{ fby } e_2]\!]_n = \begin{cases} [\![e_1]\!]_0 & \text{si } n = 0 \\ [\![e_2]\!]_{n-1} & \text{si } n > 0 \end{cases}$$

Informellement, `x fby y` calcule le flot obtenu en insérant le premier élément du flot `x` devant tous les éléments du flot `y`. En Heptagon, cet opérateur associe à droite : l'expression `x fby y fby z` est un raccourci pour `x fby (y fby z)`.

Question 4. *Expliquer pourquoi choisir de rendre l'opérateur* **`fby`** *associatif à gauche serait bien moins utile.*

On peut illustrer le fonctionnement de **`fby`** avec, par exemple, le nœud `i` ci-dessous, variante du précédent où on choisit entre `x` et `2 * y` selon la valeur courante du flot `true fby z`, variable au cours du temps.

```
node i(k : int) returns (o : int)
var x, y : int; z : bool;
let
  (x, y, z) = f();
  o = k + if true fby z then 2 * x else y;
tel
```

On peut comprendre son comportement via un chronogramme qui représente les flots de sortie et locaux de `i` pour une entrée `k` arbitraire.

k	4	-12	27	48	21	-20	5	...
x	1	1	1	1	1	1	1	...
y	42	42	42	42	42	42	42	...
z	*false*	*false*	*false*	*false*	*false*	*false*	*false*	...
`true fby z`	*true*	*false*	*false*	*false*	*false*	*false*	*false*	...
o	6	30	69	90	63	22	47	...

On peut ainsi vérifier que $[\![\mathtt{o}]\!]_0 = [\![\mathtt{k}]\!]_0 + 2$ et $[\![\mathtt{o}]\!]_{n+1} = [\![\mathtt{k}]\!]_{n+1} + 42$.

L'opérateur **fby** trouve toute son utilité en conjonction avec l'utilisation de définitions récursives. Pour vous en convaincre, essayez de résoudre la question suivante.

Question 5. *Définir un nœud* `half` *avec une seule sortie booléenne qui calcule le flot booléen périodique* `o` *alternant entre true et false, c'est à dire tel que* $[\![\mathtt{o}]\!]_{2k} = \mathit{true}$ *et* $[\![\mathtt{o}]\!]_{2k+1} = \mathit{false}$*. Les premières valeurs du flot* `o` *doivent donc être true, false, true, false . . .*

Pour résoudre cette question, on peut observer que la première valeur de `o` doit être le booléen *true*, suivi de la négation du flot `o` lui même ! Ce qui nous mène à la définition suivante.

```
node half() returns (o : bool)
let
  o = true fby not o;
tel
```

Pour nous convaincre du fonctionnement, on peut effectuer un bref calcul : on a $[\![\mathtt{o}]\!]_0 = \mathit{true}$ et $[\![\mathtt{o}]\!]_{n+1} = \overline{[\![\mathtt{o}]\!]}_n$, où $\overline{b}$ désigne la négation d'un booléen b. On peut aussi représenter les flots `o` et `not o` sur un chronogramme, et observer que le flot `not o` est égal au suffixe de `o` qui commence au deuxième instant.

`o = true fby not o`	*true*	*false*	*true*	*false*	*true*	*false*	...
`not o`	*false*	*true*	*false*	*true*	*false*	*true*	...

On a vu à la fin de la séance précédente que les définitions récursives peuvent introduire des cycles problématiques. Ce n'est pas le cas de la définition de `o` dans le nœud `i`, qui est parfaitement *causale*. En effet, on voit qu'en dépliant la définition de `o` suffisamment, on peut obtenir un nombre de valeurs arbitraire :

$$
\begin{aligned}
[\![\mathtt{o}]\!] &= [\![\mathtt{true\ fby\ not\ o}]\!] \\
&= \mathit{true} :: [\![\mathtt{not\ (true\ fby\ not\ o)}]\!] \\
&= \mathit{true} :: \mathit{false} :: [\![\mathtt{not\ (not\ o)}]\!] \\
&= \mathit{true} :: \mathit{false} :: [\![\mathtt{o}]\!] \\
&= \mathit{true} :: \mathit{false} :: \mathit{true} :: [\![\mathtt{not\ o}]\!] \\
&\ldots
\end{aligned}
$$

On écrit $x :: xs$ pour le flot dont la tête est le scalaire est x et la queue est le flot xs. Il s'agit d'une opération qui n'est pas disponible telle quelle en Heptagon. On a

$$[\![e_1 \ \mathtt{fby}\ e_2]\!] = [\![e_1]\!]_0 :: [\![e_2]\!].$$

On peut écrire beaucoup de programmes intéressants en combinant l'opérateur **fby** et des définitions récursives. En particulier, les définitions mutuellement récursives. L'exemple suivant montre comment les utiliser pour définir simultanément le flot `nat` des entiers naturels et celui des entiers strictement positifs `pos`.

`nat = 0 fby pos`	0	1	2	...
`pos = nat + 1`	1	2	3	...

```
node j() returns (nat, pos : int)
let
  nat = 0 fby pos;
  pos = nat + 1;
tel
```

Encore une fois, cette définition est causale : les flots `nat` et `pos` définissent tous deux une infinité d'éléments.

Question 6. *Dépliez les définitions de `nat` et `pos` pour montrer qu'ils contiennent au moins trois éléments chacuns, à la manière de ce que nous avons fait pour la sortie du nœud `half`.*

Si l'opérateur **fby** est le plus important, deux autres opérateurs sont également utiles : il s'agit de l'opérateur unaire **pre** et de l'opérateur binaire **->** (prononcer "*init*"). Leur sémantique est décrite par les équations suivantes.

Alternativement, on pourrait définir $[\![\textbf{pre}\ e]\!] = nil :: [\![e]\!]$.

$$[\![\textbf{pre}\ e]\!]_n = \begin{cases} nil & \text{si } n = 0 \\ [\![e]\!]_{n-1} & \text{si } n > 0 \end{cases} \qquad [\![e_1\ \textbf{->}\ e_2]\!]_n = \begin{cases} [\![e_1]\!]_0 & \text{si } n = 0 \\ [\![e_2]\!]_n & \text{si } n > 0 \end{cases}$$

La sémantique de **pre** exige une explication. En Heptagon, on suppose que chaque flot est capable de transporter une valeur spéciale baptisée *nil*, qui représente un flot non initialisé. C'est la valeur produite par l'opérateur **pre** au premier instant. Elle est absorbante par tous les opérations arithmétiques et logiques — par exemple, $nil + x = x + nil = nil$. Le compilateur Heptagon utilise une *analyse d'initialisation* pour s'assurer que cette valeur n'influe pas sur les résultats du calcul. Ainsi, le nœud ci-dessous est rejeté puisque sa sortie n'est pas initialisée au premier instant.

```
node i(x : int) returns (y : int)
let
  y = pre x;
tel
```

Certains programmeurs préfèrent utiliser **pre** et **->** à **fby** dans la mesure où leur emploi permet de séparer proprement, lors de la définition d'un flot x, le cas de base x_0 du cas inductif x_{n+1}. Ainsi, pour définir le flot `nat`, on peut partir de l'équation intuitive `nat = 1 + pre nat`, puis l'initialiser via l'opérateur **->** comme suit.

```
node k() returns (nat : int)
let
  nat = 0 -> (1 + pre nat);
tel
```

Question 7. *Pouvez-vous exprimer* `fby` *en utilisant uniquement* `pre` *et* `->` *?*

S'il peut être tentant de remplacer systématiquement `fby` par l'usage conjoint de `pre` et `->`, il s'avère plus naturel dans certaines situations. Par exemple, pour donner une définition simultanée de `nat` et `pos` comme vu précédemment.

Causalité. Les exemples précédents montrent l'importance des définitions récursives dans la programmation synchrone à flots de données. Néanmoins, la récursion est aussi utile que dangereuse : on a vu qu'il est facile d'écrire des cercles vicieux, comme `x = x`. On peut envisager ces équations de deux manières.

1. On peut décider que n'importe quelle suite en est solution, auquel cas le langage devient non-déterministe.
2. On peut décider qu'elles n'ont pas de contenu calculatoire, c'est à dire qu'on ne peut jamais obtenir le premier élément de `x` simplement en dépliant l'équation. Elles ne sont pas *causales*.

En Heptagon, et dans ce cours, on va opter pour la seconde option, et préserver le déterminisme du langage en rejetant ces cercles vicieux [17]. Quel critère algorithmique employer pour rejeter, tout en acceptant la récursion mutuelle de codes tels que le nœud `i` défini plus haut ? La recherche en langages synchrones a proposé de nombreuses solutions à ce problème. Heptagon utilise une des plus simples d'entre elles : **les dépendances cycliques *instantanées* sont interdites**. On peut comprendre ce critère en dessinant le *graphe de dépendance* d'un nœud. Il s'agit d'un graphe orienté dont les sommets x, y sont les variables déclarées dans le nœud et les arcs $x \to y$ indique que y dépend de x. La notion de dépendance est très simple : y dépend de x si x apparaît dans la définition de y. En particulier, une entrée ne peut jamais dépendre d'aucune variable puisqu'elle n'a pas de définition dans le corps du nœud. On distingue, de plus, les dépendances instantanées des dépendances *retardées*. Ces dernières sont celles où x apparaît dans la définition de y soit dans l'argument gauche d'une utilisation de l'opérateur `fby`, ou dans l'argument d'une utilisation de l'opérateur `pre`. Ainsi, le nœud `i` est causal parce que `nat` dépend de `pos` de façon retardée, et donc qu'aucun cycle instantané n'est présent (cf. figure 10, partie supérieure). En revanche, si l'on utilise `->` à la place de `fby` dans la définition de `nat`, le graphe de dépendances devient cyclique (cf. figure 10, partie inférieure). En d'autres termes, le flot `nat2` dépend instantanément de lui même (à travers `pos2`).

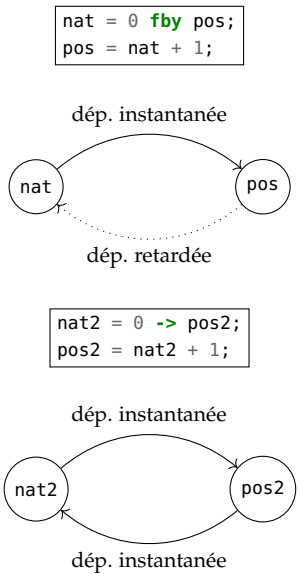


FIGURE 10: dépendances et causalité.

17. D'autres langages synchrones comme Signal optent pour le premier point de vue. Les programmes écrits dans ces langages décrivent donc des *relations* plutôt que des *fonctions*. On peut lire l'article de Le Guernic et al. [10] pour en apprendre plus sur cette approche.

Question 8. *Essayez de développer* ⟦`nat2`⟧. *Que constatez-vous ?*

Si vous essayez de compiler le bloc d'équation définissant `nat2` et `pos2`, vous obtiendrez un message d'erreur analogue à celui donné au tout début de cette section.

```
Causality error: the following constraint is not causal.
^pos2 < nat2 || ^nat2 < pos2
```

La formule, ou *contrainte*, qui accompagne ce message est une représentation compacte du graphe de dépendance de ce nœud. La première sous-contrainte `^pos2 < nat2` indique que la *lecture* de `pos2`, representée par le préfixe `^`, doit avoir lieu strictement avant l'*écriture* de `nat2`, puisque ce dernier en dépend. La deuxième, `^nat2 < pos2`, indique que la lecture de `nat2` doit avoir lieu strictement avant l'écriture de `pos2`. L'opérateur `||` indique que ces deux contraintes sont vraies *en parallèle*, c'est à dire simultanément. De plus, toute variable `x` induit une contrainte implicite `x < ^x` (elle doit avoir été écrite avant d'être lue). On a donc la contrainte totale

```
^pos2 < nat2 || ^nat2 < pos2 || pos2 < ^pos2 || nat2 < ^nat2
```

qui n'est pas satisfiable, puisque la transitivité de l'ordre implique que `nat2 < nat2` et `pos2 < pos2`. Ce raisonnement est l'analogue symbolique de l'existence d'un cycle dans le graphe de dépendances.

Réinitialisation. On a vu avec les exemples précédents qu'Heptagon disposait de trois opérateurs *séquentiels* primitifs : **`fby`**, **`pre`** et **`->`**. Ces opérateurs élémentaires peuvent être combinés pour décrire des comportements temporels complexes. L'ensemble des opérateurs séquentiels utilisés dans un nœud `f`, ainsi que dans les nœuds appelés depuis `f`, détermine l'*état* de `f`. C'est l'état du transducteur généré par le compilateur `heptc` à partir de la fonction de flot écrite par le programmeur.

Il peut, dans certaines circonstances, être utile de réinitialiser l'état d'un nœud, ou plus généralement d'un fragment de code. Pour ce faire, Heptagon dispose d'une construction particulière, dite de *réinitialisation modulaire*. Contrairement aux constructions vues jusqu'à présent, elle ne s'applique pas à une expression — ce n'est pas un opérateur — mais à un *bloc* de définitions. Les valeurs calculées par **`reset`** `block` **`every`** `c` sont celles calculées par le bloc `block`, mais l'état interne de celui-ci est réinitialisé lorsque la valeur courante du flot booléen `c` est vrai. Ainsi, le code ci-dessous réinitialise le calcul des entiers naturels dès que le flot local `c` prend la valeur *true*.

```
node nat_reset() returns (o : int)
var c : bool;
let
```

```
  reset o = 0 fby (o + 1); every c;
  c = true fby false fby false fby c;
tel
```

Ainsi, le nœud `k` ci-dessus calcule une suite d'entiers ultimement périodique, comme le montre le chronogramme suivant.

o	0	1	2	0	1	2	0	1	2	...
c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	...

La construction de réinitialisation modulaire peut rendre un programme difficile à comprendre, et est donc à utiliser avec parcimonie. On verra qu'elle est surtout utilisée par la mécanique interne des compilateurs synchrones, comme base pour des constructions de plus haut niveau.

Types structurés et énumérés. Heptagon permet la définition de types structurés : types énumérés, types enregistrements, tableaux. Les types enregistement ressemblent aux enregistrements d'OCaml, ou encore aux `structs` du langage C. Les types énumérés sont des types finis dont chaque élément a un nom déclaré. Nous étudierons les tableaux ultérieurement.

Les types structurés sont utilisables pour manipuler plusieurs valeurs simultanément, en donnant un nom à chacune d'entre elles.

```
type cpl = { re : float; im : float }

fun add(x, y : cpl) returns (o : cpl)
let
  o = { re = x.re +. y.re; im = x.im +. y.im }
tel

fun conj(x : cpl) returns (o : cpl)
let
  o = { re = x.re; im = -. x.im }
tel
```

Le code ci-dessous fournit un exemple d'utilisation d'un enregistrement pour manipuler des couples de flottants représentant des nombreux complexes. Le champ transportant la partie réelle est `re` et celui transportant la partie imaginaire est `im`. Comme pour les types scalaires, un type enregistrement décrit des flots d'enregistrements, et un enregistrement littéral tel que `{ re = 1.0; im = 0.0 }` représente un flot constant.

Un exemple d'utilisation des types énumérés est donné par le programme ci-dessous, qui encode un automate fini reconnaissant le langage rationnel $(ab^*c)^+$ (cf. figure 11). On utilise les types énumérés pour définir le type des lettres de l'alphabet d'entrée, ainsi que le type

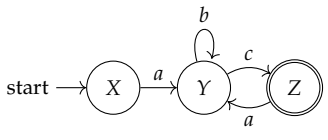


FIGURE 11: automate pour $(ab^*c)^+$.

des états. Celui-ci comprend, en plus des états X, Y, Z, un état `Dead` représentant l'état puits implicite dans les automates incomplets.

```
(* Type des lettres de l'alphabet d'entrée. *)
type alpha = A | B | C

(* Type des états de l'automate. *)
type astate = X | Y | Z | Dead

(* Automate reconnaissant le langage (a b* c)+. *)
node j(l : alpha) returns (accept : bool)
var s, sprev : astate;
let
  s = if (sprev, l) = (X, A) then Y
      else if (sprev, l) = (Y, B) then Y
      else if (sprev, l) = (Y, C) then Z
      else if (sprev, l) = (Z, A) then Y
      else Dead;
  sprev = X fby s;
  accept = (s = Z);
tel
```

Question 9. *Dessinez un chronogramme représentant les six premières valeurs des flots `l`, `s`, `sprev` et `accept` et où les six premières valeurs de `l` sont $a, b, b, c, a, b, c, c, a, b, c$.*

Horloges

Entrelacement. Jusqu'ici, nous n'avons écrit que des exemples où chaque flot avance au même rythme. Cette contrainte semble naturelle, dans la mesure où l'on traite de fonctions synchrones. Néanmoins, les langages synchrones comme Heptagon ou SCADE offrent une flexibilité utile en pratique.

Question 10. *Définissez le flot `o` tel que $[\![o]\!]_{2k} = k$ et $[\![o]\!]_{2k+1} = 0$.*

Une réponse à cette question est fournie par le nœud ci-dessous. Celui-ci définit le flot `x` des entiers naturels qui "bégaie", c'est-à-dire, où chaque entier est répété deux fois. On utilise ensuite l'opérateur **`if`** pour remplacer tous les éléments de rang impair par des 0.

```
node k() returns (o : int)
var x : int; h : bool;
let
  x = 0 fby 0 fby (x + 1);
  o = if h then x else 0;
  h = true fby not h;
tel
```

h	*true*	*false*	*true*	*false*	...
x	0	0	1	1	...
o	0	0	1	0	...

Le code ci-dessus est critiquable : passer par la définition du flot `x` est peu naturel. Peut-on réutiliser la définition du flot des entiers naturels `nat = 0 fby (nat + 1)` donnée précédemment ?

Question 11. *Montrez que remplacer `x` par `nat` dans `k` définit un flot `o` tel que* $[\![o]\!]_{2k} = 2k$ *et* $[\![o]\!]_{2k+1} = 0$.

Ce qu'on voudrait faire, c'est *entrelacer* les valeurs du flot `nat` avec celles du flot `0`. Heptagon dispose d'une construction idoine, l'opérateur d'entrelacement `merge c e1 e2`, qu'on peut comprendre comme un cousin assez lointain de `if c then e1 else e2`. Contrairement à la conditionnelle, la fusion de flots n'est pas un opérateur point-à-point. Informellement, lorsque le prochain élément de $[\![c]\!]$ est vrai (resp. faux), la sortie de $[\![$`merge c e1 e2`$]\!]$ est produite en consommant un élement de $[\![e1]\!]$ (resp. $[\![e2]\!]$), sans consommer celui de $[\![e2]\!]$ (resp. $[\![e1]\!]$) — contrairement à ce qui se passerait avec $[\![$`if c then e1 else e2`$]\!]$.

Avant de donner une définition plus rigoureuse du comportement de l'opérateur d'entrelacement, il est utile de revenir à notre exemple, reformulé avec l'aide de l'entrelacement pour définir `o` en fonction du flot des entiers naturels sans bégaiement.

```
node l() returns (o : int)
var x : int; h : bool;
let
  x = 0 fby (x + 1);
  o = merge h x 0;
  h = true fby not h;
tel
```

Pour comprendre le fonctionnement de ce nœud, on peut suivre la même méthodologie que précédemment, et dessiner un chronogramme. Pour ce faire, il faut décider comment les valeurs de chaque flot sont calculées au cours du temps. Une première possibilité naïve serait de supposer que `nat` et `h` sont calculés au même rythme, comme représenté par le chronogramme ci-dessous.

h	*true*	*false*	*true*	*false*	*true*	*false*	*true*	*false*	*true*	...
x	0	1	2	3	4	5	6	7	8	...
o	0	0	1	0	2	0	3	0	4	...

Sur ce chronogramme, on a représenté en rouge les dépendances entre les valeurs du flot `x` et celles du flot `o`. Chacune de ces valeurs, à l'exception la première, est consommée *strictement* après avoir été produite. Elle doit donc être mémorisée. Plus précisément, à l'instant $2k$, le flot $[\![o]\!]$ ne contient que les k premières valeurs de $[\![x]\!]$, et les k valeurs suivantes doivent être mémorisées. En d'autres termes, la quantité de mémoire nécessaire pour exécuter semble croitre irrémédiablement en

fonction du temps. C'est inadmissible : du point de vue de SCADE et Heptagon, **un programme synchrone doit s'exécuter en mémoire bornée**.

Question 12. *Quelle implémentation du nœud* `l` *pourrait garantir cette propriété ?*

Pour exécuter ce programme en espace borné, il faut nécessairement que le flot `x` soit produit *plus lentement* que le flot `o`, de sorte que ses éléments soient disponibles *juste à temps*. C'est ce que le chronogramme ci-dessous représente, les espaces vides marquant les instants auxquels le prochain élément du flot correspondant n'est pas calculé.

h	*true*	*false*	*true*	*false*	*true*	*false*	*true*	*false*	*true*	...
x	0		1		2		3		4	...
o	0	0	1	0	2	0	3	0	4	...

On voit qu'en ralentissant `x`, on a réussi à aligner la production de ses éléments avec leur consommation dans le flot `o`. Mémoriser les valeurs de `x` n'est donc plus nécessaire.

On peut maintenant donner un sens précis à l'opérateur d'entrelacement. Sur le chronogramme précédent, on a marqué l'absence d'un flot par une case vide. En pratique, pour décrire l'opérateur d'entrelacement, il est utile de disposer d'une valeur spéciale symbolisant l'absence, notée *abs*, et qu'on peut aussi utiliser dans les chronogrammes.

h	*true*	*false*	*true*	*false*	*true*	*false*	*true*	*false*	*true*	...
x	0	*abs*	1	*abs*	2	*abs*	3	*abs*	4	...
o	0	0	1	0	2	0	3	0	4	...

En manipulant la valeur spéciale *abs*, on peut désormais décrire la sémantique de l'opérateur d'entrelacement.

$$[\![\texttt{merge}\ c\ e_1\ e_2]\!]_k = \begin{cases} [\![e_1]\!]_k \text{ si } [\![\texttt{c}]\!]_k = \mathit{true} \text{ et } [\![e_2]\!]_k = \mathit{abs} \\ [\![e_2]\!]_k \text{ si } [\![\texttt{c}]\!]_k = \mathit{false} \text{ et } [\![e_1]\!]_k = \mathit{abs} \end{cases}$$

On voit que cette définition est partielle. Lorsque le $k$ème élément du flot $[\![\texttt{c}]\!]$ est vrai, $[\![\texttt{merge c e1 e2}]\!]_k$ est défini si seulement si le $k$ème élément du flot $[\![\texttt{e2}]\!]$ est absent, et symétriquement. De plus, on considèrera que si $[\![\texttt{merge c e1 e2}]\!]_k$ est indéfini, alors $[\![\texttt{merge c e1 e2}]\!]_{k'}$ est également indéfini pour tout $k' > k$.

Cette formulation à l'aide de valeurs absentes permet d'en faire une fonction synchrone : les n premières valeurs de sa sortie dépendent uniquement des n premières valeurs de ses entrées. C'est cette propriété qui assure que l'opérateur d'entrelacement peut toujours être utilisé sans avoir à mémoriser implicitement ses entrées, et donc qu'il est combinatoire ! Le prix à payer est que les flots à entrelacer doivent comprendre des valeurs absentes exactement aux rangs attendus, sans

quoi l'entrelacement est indéfini. Les langages synchrones à flots de données comme Heptagon assurent cette propriété via une analyse statique dédiée, baptisée *calcul d'horloge*.

Sélection. Le fonctionnement du calcul d'horloge est plus facile à expliquer sur un opérateur qui joue un rôle inverse à celui de l'opérateur d'entrelacement. Si l'opérateur d'entrelacement permet de combiner plusieurs flots lents pour en obtenir un rapide, l'opérateur de *sélection* [18] transforme un flot rapide en flot lent par la suppression de certains de ses éléments. Intuitivement, ⟦e `when` c⟧ est le flot ⟦e⟧ dont on a conservé la valeur $[\![e]\!]_k$ ssi $[\![c]\!]_k$ est vrai. Les autres valeurs sont remplacées par *abs*, dans le but de rendre synchrone l'opérateur de sélection.

$$[\![e \texttt{ when } c]\!]_k = \begin{cases} [\![e]\!]_k \text{ si } [\![c]\!]_k = \mathit{true} \\ \mathit{abs} \text{ si } [\![c]\!]_k = \mathit{false} \end{cases}$$

18. Il est aussi appelé *opérateur d'échantillonnage* (*sampling* en anglais) dans la documentation d'Heptagon et la littérature scientifique.

Question 13. *Définissez un nœud envoyant un flot d'entiers* `y` *dans le flot* `o` *tel que* $[\![o]\!]_{2k} = k$ *et* $[\![o]\!]_{2k+1} = [\![y]\!]_{2k+1}$.

Il s'agit d'une variation sur le dernier nœud que nous ayons defini. Les valeurs de rang pair dans y doivent éliminées pour ne laisser que les valeurs de rang pair.

La construction `y when h` n'est que du sucre syntaxique pour `y when true(h)`.

```
node m(y : int) returns (o : int)
var x : int; h : bool;
let
  x = 0 fby (x + 1);
  o = merge h x (y when false(h));
  h = true fby not h;
tel
```

On peut dessiner un chronogramme pour ce nœud où les valeurs de y sont laissées abstraites.

y	y_0	y_1	y_2	y_3	y_4	y_5	y_6	...
h	*true*	*false*	*true*	*false*	*true*	*false*	*true*	...
x	0	*abs*	1	*abs*	2	*abs*	3	...
y `when false(h)`	*abs*	y_1	*abs*	y_3	*abs*	y_5	*abs*	...
o	0	y_1	1	y_3	2	y_5	4	

On constate bien que les flots ⟦x⟧ et ⟦y `when false(h)`⟧ ne sont jamais présents au même instant, ce qui est nécessaire au bon fonctionnement de l'opérateur d'entrelacement avec la sémantique donnée plus haut. Cette condition est **vérifiée statiquement par le compilateur** via le calcul d'horloge. Pour vous en convaincre, essayez de compiler le nœud précédent, en remplaçant `y when false(h)` par `y when h`.

```
$ heptc badmerge.ept
(y when h) : File "badmerge.ept", line 5, characters 17-25:
>  o = merge h x (y when h);
>                 ^^^^^^^^
Clock Clash: this value has clock 'a on true(h),
but is expected to have clock 'a on false(h).
```

Ce message indique que l'expression `y when h` n'a pas la bonne *horloge*. L'*horloge* d'une expression e est une formule qui décrit un flot booléen *ck* tel que ck_k est vrai ssi $[\![e]\!]_k \neq abs$. Elle permet de s'assurer que les valeurs transportées par le flot sont cohérentes avec son utilisation. Ce n'est pas le cas dans l'exemple qui nous occupe : le flot `e when h` a l'horloge `'a on true(h)`, indiquant qu'il est présent lorsque `h` est vrai, mais devrait avoir une horloge de la forme `'a on false(h)`, puisqu'en tant que troisième argument de `merge h` il est lu lorsque `h` est faux.

Heptagon emploie un jeu de règles pour décider de la cohérence des horloges d'un programme. Nous n'allons pas rentrer dans les détails du fonctionnement de ce système. Nous nous contenterons de préciser que les horloges peuvent être vues comme des types, et le calcul d'horloge comme un système de types. Quelques-unes de ses règles sont présentées à la figure 12.

$$\frac{\Gamma \vdash e_1 : C \qquad \Gamma \vdash e_2 : C}{\Gamma \vdash e_1 + e_2 : C}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : C \\ \Gamma \vdash e_1 : C \texttt{ on } \texttt{true}(x) \\ \Gamma \vdash e_2 : C \texttt{ on } \texttt{false}(x)\end{array}}{\Gamma \vdash \texttt{merge } x\ e_1\ e_2 : C}$$

$$\frac{\Gamma \vdash e : C \qquad \Gamma \vdash x : C}{\Gamma \vdash e \texttt{ when } b(x) : C \texttt{ on } b(x)}$$

FIGURE 12: calcul d'horloge (extrait).

— La première spécifie que les deux arguments d'une addition doivent avoir la même horloge, qui est également l'horloge du résultat.

— La seconde spécifie que le deuxième argument de `merge` doit être présent lorsque la condition est vraie, et le second lorsque la condition est fausse.

— Le troisième indique que les deux arguments de `when` doivent être présents aux mêmes instants mais que la sortie est présente lorsque, en plus, la condition est vraie.

On peut illustrer la première des trois règles avec un exemple : le programme ci-dessous est mal typé car les deux arguments de l'addition ne sont pas présents aux mêmes instants.

```
node n(x : int) returns (o : int)
var h : bool;
let
  h = true fby not h;
  o = x + (x when h);
tel
```

```
(x when h) : File "badplus.ept", line 5, characters 11-19:
>  o = x + (x when h);
>           ^^^^^^^^
Clock Clash: this value has clock 'a on true(h),
but is expected to have clock 'a.
```

Pour terminer notre discussion des horloges, on peut présenter un exemple d'utilisation moins artificiel que les précédents.

Un nœud très utile dans les programmes synchrone est l'intégrateur, qui calcule une approximation numérique de l'intégrale de son flot d'entrée. Le schéma d'intégration dit d'Euler explicite, implémenté dans le nœud `itgr` ci-dessous, est sans doute le plus simple à programmer.

```
node itgr(x, h, ini : float) returns (o : float)
let
  o = (ini fby o) +. h *. x;
tel
```

L'entrée `x` est le flot à intégrer, le flot `ini` la valeur initiale de l'intégrateur, et le flot `h` donne le pas d'intégration, qu'on peut comprendre comme le temps physique écoulé depuis l'instant logique précédent. Si `h` est suffisamment petit, les valeurs successives de `y` offrent de bonnes approximations de l'intégrale de Riemann de `x`.

Imaginons maintenant qu'on veuille rajouter une entrée supplémentaire à l'intégrateur : un flot booléen `en` qui contrôle à quels instants `x` doit être intégré. Lorsque `en` est faux, la sortie doit conserver la valeur qu'elle avait à l'instant précédent. Ce type d'intégrateur est très utile. On peut programmer cet intégrateur interruptible en combinant tous les opérateurs vus juqu'à présent : entrelacement, sélection et opérateurs séquentiels.

```
node itgr_enable(x, h, ini : float; en : bool)
        returns (o : float)
var oi : float;
let
  oi = itgr(x when en, h when en, ini when en);
  o = merge en oi ((ini fby o) when false(en));
tel
```

Un point important est qu'en Heptagon, les nœuds sont automatiquement *polymorphes* vis-à-vis des horloges : on peut appliquer `itgr` à des arguments sur une horloge quelconque, du moment que cette horloge est la même pour les trois arguments.

Question 14. *Donnez un chronogramme pour ce nœud, en prenant comme premières valeurs : pour le flot* `en`*, les valeurs* true, true, false, true, false*; pour* `ini`*, le flot constant* 0.0*; pour* `h` *le flot constant* 0.2*; pour le flot* `x`*, des valeurs abstraites* x_0, x_1, x_2, x_3, x_4.

Question 15. *Que se passe-t-il si on remplace l'équation pour* `oi` *dans le nœud* `itgr_enable` *par* `oi = itgr(x, h, ini) when en`*? Pouvez-vous expliquer ce résultat par un chronogramme ?*

Automates

Les constructions manipulant les horloges, **merge** et **when**, sont des ingrédients essentiels aux langages comme Heptagon et SCADE. On a toutefois vu que leur maniement est un peu délicat, puisqu'il exige de comprendre un tant soit peu le fonctionnement du calcul d'horloge. En pratique, les programmeurs utilisent plutôt des constructions de *contrôle*, qui permettent d'activer et désactiver des blocs d'équation de diverses manières. Heptagon et SCADE réduisent ensuite ces constructions à celles sur les horloges durant le processus de compilation. Nous allons maintenant discuter de ces constructions de contrôle, qui permettent notamment la programmation directe d'automates.

Pour notre premier exemple de construction de contrôle, nous allons réimplémenter le nœud `j`, qui reconnaît le langage $(ab^*c)^+$.

```
type alpha = A | B | C

node p(l : alpha) returns (accept : bool)
let
  automaton
    state X
      do accept = false
         unless l = A then Y | true then Dead

    state Y
      do accept = false
         unless l = B then Y | l = C then Z | true then Dead

    state Z
      do accept = true
         unless l = A then Y | true then Dead

    state Dead
      do accept = false
  end
tel
```

Son interface n'a pas changé. En revanche, son corps n'est pas formé directement d'un ensemble d'équations, mais d'un automate, introduit par le mot-clef **automaton**. Un automate doit spécifier une liste d'états, introduits par le mot-clef **state**, le premier d'entre eux étant par convention l'état initial [19] de l'automate. Chaque état a un nom qui doit commencer par une majuscule. Chaque état spécifie un bloc d'équations après le mot-clef **do**, et éventuellement une liste de transitions. Seules les équations de l'état courant de l'automate sont actives et dictent la

19. Contrairement aux automates étudiés dans les cours de langages formels, les automates d'Heptagon n'ont pas à proprement parler d'état final, puisqu'ils ne reconnaissent pas un langage mais contrôlent quelles équations sont actives à quel instant.

valeur des variables correspondantes. Par exemple, une transition

```
unless c1 then S1 | c2 then S2 | ... | cN then SN
```

spécifie que si la condition `c1` s'évalue à vrai, le prochain état est `S1` ; sinon, on évalue `c2`, et le prochain état devient `S2` si cette condition s'évalue à vrai ; sinon, on évalue `c3`, et ainsi de suite. Si aucune condition n'est vraie, on reste dans l'état courant. Il existe des transitions de diverses sortes, que nous allons décrire tout de suite.

Transitions fortes et faibles. Comme on l'a vu, chaque état d'un automate peut contenir plusieurs transitions de sortie, contrôlées par des conditions booléennes. Dans l'exemple précédent, on a utilisé des transitions de type `unless`, dites *transitions fortes*. Les conditions des transitions fortes sont **testées au début de l'instant courant**, l'état correspondant n'est donc pas activé si l'une d'entre elles est vraie. Ainsi, le nœud `a0` ci-dessous produit le flot constant `true` car l'équation `o = false` n'est jamais active — il entre dans l'état `B` au début du premier instant.

```
node a0() returns (o : bool)
let
  automaton
    state A
      do o = false
      unless true then B
    state B
      do o = true
  end
tel
```

En plus des transitions fortes, on dispose également de transitions *faibles*. Les conditions des transitions faibles sont **testées à la fin de l'instant**, et déterminent donc l'état dans lequel l'automate commencera l'instant suivant. Ainsi, le nœud `a1` ci-dessous produit le flot ⟦`false fby true`⟧, car la condition est testée à la fin du premier instant — l'état `B` est donc actif à partir du deuxième instant.

```
node a1() returns (o : bool)
let
  automaton
    state A
      do o = false
      until true then B
    state B
      do o = true
  end
tel
```

Comment choisir entre transitions faibles et fortes ? Il existe une différence importante entre ces deux types de transitions du point de vue de la causalité. Les conditions des transitions fortes étant testées au début de l'instant, les variables définies dans le corps de l'état sortant dépendent instantanément de celles lues dans les conditions. Pour cette raison, la variante des nœuds ci-dessous n'est pas causale.

```
node a2() returns (o : bool)
let
  automaton
    state A
      do o = false
      unless not o then B
    state B
      do o = true
  end
tel
```

Dans ce programme, la définition de `o` dans l'état `A` dépend instantanément de la valeur de `o`, puisque savoir si elle est active exige de tester la condition `not o` de la transition forte, condition qui dépend instantanément de `o`. Le compilateur Heptagon illustre ce fait via le message d'erreur suivant.

```
Causality error: the following constraint is not causal.
^o < o || o
```

Ce problème ne se pose plus si l'on remplace la transition forte par une transition faible, comme ci-dessous. La condition `not o` étant testée à la fin de l'instant, la définition de `o` n'en dépend que de façon retardée, et ce programme n'a pas de problème de causalité. Le résultat est équivalent à `a1`.

```
node a3() returns (o : bool)
let
  automaton
    state A
      do o = false
      until not o then B
    state B
      do o = true
  end
tel
```

En règle générale, les programmeurs synchrones ont tendance à préférer par défaut les transitions faibles aux transitions fortes, puisqu'elles évitent d'avoir à se soucier de certaines boucles de causalité. Certains

exemples, néanmoins, sont plus naturels avec des transitions fortes, comme l'automate reconnaissant le langage régulier $(ab^*c)^+$.

En Heptagon, un automate ne peut effectuer qu'un seul changement d'état par instant dans la plupart des cas : la sémantique du langage interdit de quitter un état par une transition forte pour entrer dans un nouvel état et en sortir immédiatement par une autre transition forte. Par exemple, le nœud `a5` ci-dessous produit le flot $[\![2\ \texttt{fby}\ 3]\!]$ car l'état `A` n'est jamais actif, mais qu'on ne peut pas sortir immédiatement de l'état `B` par une transition forte alors qu'on vient d'y entrer au premier instant.

```
node a4() returns (o : int)
let
  automaton
    state A
      do o = 1
      unless true then B
    state B
      do o = 2
      unless true then C
    state C
      do o = 3
  end
tel
```

Il existe une seule exception à cette règle, où un automate peut effectuer deux transitions en un instant, et donc passer par un état transitoire qui n'est jamais actif. Elle se produit lorsqu'on sort d'un état `A` par une transition faible à la fin de l'instant n pour entrer à l'instant $n+1$ dans un état `B` qui dispose d'une transition forte dont la condition est vraie et qui mène à un état `C`. Dans ce cas, les définitions de l'état `B` ne sont jamais actives, et l'automate passe directement de l'état `A` à l'état `C`.

```
node a5() returns (o : int)
let
  automaton
    state A
      do o = 1
      until true then B
    state B
      do o = 2
      unless true then C
    state C
      do o = 3
  end
```

```
tel
```

Le nœud `a5` ci-dessus illustre ce comportement : à la fin du premier instant, il passe à l'état `B`, mais en sort au début du deuxième instant pour entrer directement dans l'état `C`. Il produit donc le flot $[\![$`1 fby 3`$]\!]$. Programmer principalement avec des transitions faibles évite d'avoir à se préoccuper des enchaînements faible/forte.

Transitions réinitialisantes et continuantes. Le nœud suivant comprend un automate à deux états qui passe de l'état initial `A` à l'état `B` après trois instants via une transition faible, avant de repasser à l'état `A` un instant plus tard. Comment évolue son flot de sortie ?

```
node f0() returns (o : int)
let
  automaton
    state A
      do o = 0 fby (o + 1)
      until o >= 3 then B
    state B
      do o = 42
      until true then A
  end
tel
```

La sortie de `f0` est périodique, comme le montre ce chronogramme.

o	0	1	2	3	42	0	1	2	3	42	0	1	2	3	42	...

Ce comportement est causé par les transitions de la forme `c then S` qui, qu'elles soient faibles ou fortes, réinitialisent leur état d'arrivée. On dit que de telles transitions sont *réinitialisantes*. Autrement dit, le flot des entiers naturels définis dans l'état `A` est réinitialisé lors de la transition qui y entre depuis `B`. De la même manière, le nœud suivant produit la suite périodique $(0\,1\,2\,3)^\omega$.

La notation u^ω, où u est un mot fini, désigne le flot périodique formé par la répétition continuelle du mot u.

```
node f1() returns (o : int)
let
  automaton
    state A
      do o = 0 fby (o + 1)
      until o >= 3 then A
  end
tel
```

Le fait qu'un état soit réinitialisé est un comportement souvent commode lors de l'écriture d'un programme réactif. Il permet de comprendre l'évolution d'un état `S` en isolation des autres, sans avoir à

considérer les transitions qui mènent à `S`. Cependant, il est parfois commode de ne pas réinitialiser un état. Pour cette raison, Heptagon offre un autre type de transition, les transitions *continuantes*. Celles-ci, notées `c continue S`, permettent d'entrer dans l'état `S` sans le réinitialiser. Elles peuvent être fortes ou faibles, tout comme les transitions réinitialisantes.

Question 16. *Quel est le flot produit par le nœud `f2` ci-dessous ?*

```
node f2() returns (o : int)
let
  automaton
    state A
      do o = 0 fby (o + 1)
      until o >= 3 then B
    state B
      do o = 42
      until true continue A
  end
tel
```

Question 17. *Comment simplifier le nœud `f3` ci-dessous ?*

```
node f3() returns (o : int)
let
  automaton
    state A
      do o = 0 fby (o + 1)
      until o >= 3 continue A
  end
tel
```

Mémoire partagée. Les constructions relatives aux automates vues jusqu'ici permettent de programmer rapidement des structures de contrôle complexes. Il leur manque toutefois un ingrédient essentiel. Pour s'en convaincre, essayez d'implémenter la spécification suivante.

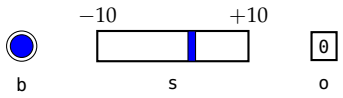


FIGURE 13: interface du nœud **`switch`**3.

Question 18. *Programmez un nœud Heptagon **`switch`**3 ayant :*

— *une entrée booléenne `b` représentant les pressions sur un bouton de commande ;*

— *une entrée entière `s` représentant la position d'un curseur, qu'on supposera comprise entre* −10 *et* 10 *;*

— *une sortie entière `o`, initialisée à* 0*, qu'on imaginera affichée sur un écran.*

*La figure 13 présente une vision figurative d'une hypothétique interface graphique comprenant bouton, curseur, et afficheur. De plus, le nœud **`switch`**3 doit obéir à la spécification suivante :*

— *il commence son exécution dans un état où* `o` *n'évolue pas ;*

— *après une pression sur le bouton de commande, il passe dans un état où* `o` *est incrémenté de la position courante du curseur ;*

— *après une nouvelle pression sur le bouton de commande, il passe dans un état où* `o` *est multiplié par la position courante du curseur ;*

— *Enfin, une dernière pression le ramène dans l'état initial.*

On pourrait essayer de programmer un tel automate avec trois états connectés par des transitions continuantes, comme suit.

```
node switch3bad(b : bool; s : int) returns (o : int)
let
  automaton
    state Idle
      do o = 0 -> pre o
      unless b continue Increment
    state Increment
      do o = (0 -> pre o) + s
      unless b continue Multiply
    state Multiply
      do o = (0 -> pre o) * s
      unless b continue Idle
  end;
tel
```

Toutefois, le comportement de cet automate n'est pas le bon, comme en témoigne le chronogramme ci-dessous.

b	1	0	0	1	0	1	0	1	0	0	0	1	0	0	...
s	1	2	3	1	5	5	2	2	1	1	2	2	0	5	...
o	1	3	6	-1	-6	0	0	8	9	10	12	-8	-8	-13	...

Ce chronogramme montre que l'opérateur `pre o`, utilisé dans un état `S`, fait référence à la valeur de `o` à son instant de *définition* précédent, c'est à dire, à l'instant d'activation précédent de `S`. Autrement dit, toutes les définitions de `o` sont purement locales à chaque état, et peuvent être considérées indépendamment.

Cependant, pour respecter notre spécification, nous voudrions plutôt que la sortie `o` soit une *mémoire* partagée globalement entre les différents états de l'automate. On peut obtenir un tel comportement via le mot-clef `last`. Celui-ci, utilisé dans une déclaration de variable, permet de faire de celle-ci une mémoire partagée. Utilisé dans une expression, il permet d'accéder à la valeur d'une mémoire à l'instant précédent [20]. De plus, une mémoire peut être initialisée, auquel cas elle n'a pas à recevoir de définition dans tous les états de l'automate — elle conserve implicitement sa valeur précédente en l'absence de définition.

20. Il est interdit d'utiliser `last` sur une variable qui n'a pas été déclarée comme étant une mémoire.

En faisant de la sortie `o` une mémoire partagée, on peut réécrire l'exemple précédent comme suit.

```
node switch3(b : bool; s : int) returns (last o : int = 0)
let
  automaton
    state Idle
      do
      unless b then Increment
    state Increment
      do o = last o + s
      unless b then Multiply
    state Off
      do o = last o * s
      unless b then Idle
  end
tel
```

On obtient, cette fois, le chronogramme attendu.

b	1	0	0	1	0	1	0	1	0	0	0	1	0	0	...
s	1	2	3	1	5	5	2	2	1	1	2	2	0	5	...
o	1	3	6	5	0	0	0	2	3	4	6	4	4	-1	...

En effet, la valeur `last o` lue dans l'état `On` est bien la dernière définie dans n'importe lequel des états de l'automate, que ce soit dans `On` soit dans `Off`, et similairement pour la lecture de `last o` dans l'état `Off`.

Autres constructions de contrôle. En plus des automates, Heptagon dispose de quelques autres constructions de contrôle plus légères, plus simples à utiliser lorsqu'un automate général n'est pas nécessaire.

— La construction **`if`**/**`then`**/**`else`** appliquée aux blocs permet de sélectionner entre deux blocs à activer en fonction d'une condition booléenne.

— La construction **`switch`** généralise la précédente en permettant d'activer un bloc d'équations selon la valeur d'un flot énuméré.

— Enfin, la construction **`present`** généralise la construction **`switch`** en permettant de sélectionner la branche à activer en fonction d'une série de conditions booléennes testées successivement. On peut éventuellement spécifier un bloc à activer par défaut si toutes les conditions sont fausses.

Dans les trois constructions ci-dessus, les flots permettant la sélection du bloc d'équations doivent être calculés par des expressions combinatoires. Cette condition est vérifiée par le compilateur Heptagon.

```
type color = Green | Amber | Cyan

fun color2rgb(c : color)
      returns (r, g, b : float)
let
  switch c
  | Green
    do r = 0.0; g = 1.0; b = 0.0;
  | Amber
    do r = 1.0; g = 0.75; b = 0.0;
  | Cyan
    do r = 0.0; g = 1.0; b = 1.0;
  end
tel
```

Tableaux et itérateurs

En plus des enregistrements, Heptagon offre un autre type de données : les tableaux, c'est-à-dire des séquences finies d'éléments de même type. Ceux-ci fonctionnent largement comme les tableaux des langages fonctionnels. On va maintenant détailler les constructions relatives aux tableaux, en commençant par les plus simples.

Types tableaux. Comme dans la plupart des langages de programmation typés, les tableaux d'Heptagon sont homogènes : ils contiennent des éléments du même type. Contrairement à un langage comme Java ou OCaml, ils sont d'une taille fixée à la compilation. Cette taille doit être spécifiée par une *expression statique*, c'est à dire une expression dont la valeur peut être calculée à la compilation — on décrira le fonctionnement de ces expressions plus loin dans cette section. La notation est `t^s`, où `t` est un type et `s` une expression statique, désigne les tableaux de type `t` de taille `s`. Comme tous les types d'Heptagon, ce type désigne en réalité un flot, flot qui transporte des tableaux. Ainsi, le type `int^10` désigne les flots de tableaux de taille 10 d'entiers, le type `float^15^30` désigne les flots de tableaux de taille 30 de tableaux de taille 15 de nombres flottants.

Constructions de base. La façon la plus simple de créer un nouveau tableau est de spécifier chacun de ses éléments. Le littéral $[e_1, e_2, \ldots, e_s]$ désigne le flot de tableau de taille s dont les éléments sont calculées par les expressions e_i, qui doivent par conséquent toutes avoir le même type. Si tous les éléments du tableau sont les mêmes, on peut écrire à la place `e^n`, raccourci pour $[e, \ldots, e]$ où e apparaît `n` fois.

```
node arr0() returns (o1, o2, o3 : int^3)
let
  o1 = [ 1, 2, 3 ];
  o2 = 42^3;
  o3 = o1 fby o2;
tel
```

o1	[1, 2, 3]	[1, 2, 3]	...
o2	[42, 42, 42]	[42, 42, 42]	...
o3	[1, 2, 3]	[42, 42, 42]	...

Le nœud ci-dessus produit trois sorties, toutes trois des flots de tableaux d'entiers de taille 3. Les deux premières sont des flots constants, contrairement au troisième. Le nœud ci-dessous donne un exemple plus intéressant de flot de tableaux qui évolue au cours du temps. Le n-ème tableau transporté par le flot `o` est $[n, n+1]$.

```
node arr1() returns (o : int^2)
var n : int;
let
  n = 0 fby (n + 2);
```

n	0	2	4	...
o	[0, 1]	[2, 3]	[4, 5]	...

```
  o = [ n, n + 1 ];
tel
```

Comme la plupart des langages de programmation, Heptagon permet l'accès indicé au contenu d'un tableau. Le langage étant dédié aux systèmes critiques, l'accès indicé y est plus rigide que dans des langages généralistes où un accès en dehors des bornes du tableau peut être détecté dynamiquement. On va tout d'abord voir les méthodes d'accès indicé les plus restrictives et sûres, avant d'aborder les autres.

L'expression `a[i]`, où `a` est une expression de type tableau `t^n` et `i` une expression statique, désigne le contenu de la `i`-ème case du tableau `a`. L'expression `i` étant statique, le compilateur Heptagon peut vérifier statiquement que son résultat appartient bien à l'intervalle $[0, n[$.

Bien que limités, les accès indicés constants permettent déjà d'écrire des nœuds intéressants. Par exemple, un *registre à décalage* stocke un nombre codé sur n bits, et produit périodiquement les bits de n, du bit de poids le plus fort au bit de poids le plus faible. Le nœud ci-dessous implémente un registre à décalage sur trois bits qui passe au bit suivant lorsque son entrée `sh` est vraie. L'entrée `ini` fournit l'entier à stocker initialement — sous sa forme de tableau de bits.

```
node shiftr3(ini : bool^3; sh : bool) returns (o : bool)
var mem, nxt : bool^3;
let
  mem = ini fby nxt;
  nxt = if sh then [ mem[2], mem[0], mem[1] ] else mem;
  o = nxt[2];
tel
```

Si le flot `ini` vaut `[true, false, true]` au premier instant, on peut obtenir par exemple le chronogramme suivant.

sh	0	1	0	0	1	1	...
mem	[1, 0, 1]	[1, 0, 1]	[1, 1, 0]	[1, 1, 0]	[1, 1, 0]	[0, 1, 1]	...
nxt	[1, 0, 1]	[1, 1, 0]	[1, 1, 0]	[1, 1, 0]	[0, 1, 1]	[1, 0, 1]	...
o	1	0	0	0	1	1	...

Question 19. *Pouvez-vous réimplémenter le nœud `shiftr3` en remplaçant les variables `mem` et `nxt`, de type `bool^3` par des variables de type `bool` ?*

Heptagon permet aussi d'accéder à une cellule d'un tableau dont l'indice a été spécifié par une expression dont la valeur n'est pas connue avant l'exécution. Le compilateur ne pouvant pas vérifier que cet accès est sûr à la compilation, cette construction exige de fournir une valeur par défaut qui sera utilisée en cas d'accès hors des bornes du tableau. On écrit `a.[e] default d` pour signifier qu'on veut accéder à la case

d'indice ⟦e⟧ du tableau ⟦a⟧, avec ⟦d⟧ la valeur à utiliser si ⟦e⟧ n'est pas comprise entre 0 et $s-1$, avec s la taille de a. Plus formellement,

$$⟦\texttt{a.[e] default d}⟧_n = \begin{cases} ⟦\texttt{a}⟧_n[⟦\texttt{e}⟧_n] & \text{si } 0 \leq ⟦\texttt{e}⟧_n < s \\ ⟦\texttt{d}⟧_n & \text{sinon.} \end{cases}$$

Une autre possibilité, plus rarement utilisée, est de tronquer le résultat de l'expression calculant l'indice pour le ramener dans les bornes du tableau. Cette construction s'écrit `a[>e<]`, et sa sémantique est

$$⟦\texttt{a[>e<]}⟧_n = ⟦\texttt{a}⟧_n[\min(\max(⟦\texttt{e}⟧_n, 0), s-1)].$$

Enfin, on peut également lire une *tranche* de tableau, c'est à dire un sous-tableau formé d'un nombre contigu de cellules. La tranche `a[lo .. hi]` doit être définie par deux expressions statiques `lo` et `hi`, de sorte à ce que la taille `hi - lo + 1` du sous-tableau lu soit calculable à la compilation.

En plus de lire les cellules d'un tableau, on peut également modifier ses cellules. Heptagon étant un langage fonctionnel, cette modification résulte en un nouveau tableau, laissant l'original intact. Ainsi, `[ t with [e] = v ]` construit à l'instant n un nouveau tableau identique à `t` à l'exception de la cellule d'indice ⟦e⟧$_n$ dont la valeur est remplacée par ⟦v⟧$_n$. Si ⟦e⟧$_n$ est en dehors des bornes du tableau ⟦a⟧$_n$, la modification n'a pas lieu et le résultat est identique à ⟦a⟧$_n$.

À l'aide de ces constructions, on peut écrire par exemple un *buffer circulaire* stockant des entiers. Un buffer circulaire est un composant mémorisant d'une taille fixée, disons n, qui mémorise les éléments d'un flot d'entiers pour les produire à un instant ultérieur. Celui que nous allons programmer dispose de trois entrées et d'une sortie.

— L'entrée `e` fournit un flot d'entiers dont l'élement courant doit être stocké lorsque l'entrée booléenne `w` est vraie.

— L'entrée booléenne `r` indique si la prochaine valeur du buffer a été consommée dans la sortie `o`.

En particulier, la valeur de l'entrée `e` n'est pertinente que lorsque `w` est vrai, et similairement pour `o` et `r`.

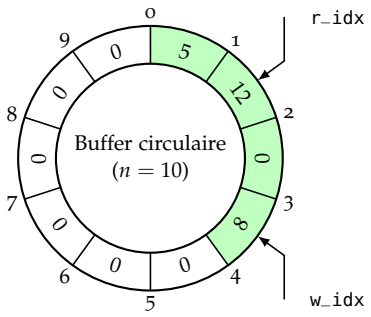


FIGURE 14: buffer circulaire à l'instant 5.

```
const n : int = 10

node ring_buffer(e : int; w, r : bool) returns (o : int)
var r_idx, w_idx : int; pa, a : int^n;
let
  o = a.[r_idx] default 0;
  pa = (0^n) fby a;
  a = if w then [ pa with [w_idx] = e ] else pa;
  r_idx = 0 fby (((if r then 1 else 0) + r_idx) % n);
```

```
  w_idx = 0 fby (((if w then 1 else 0) + w_idx) % n);
tel
```

Le code ci-dessus implémente un tel buffer circulaire en utilisant deux flots d'entiers, `w_idx` et `r_idx`, qui contiennent respectivement les prochains indices auxquels écrire et lire. Le flot de tableaux `a` transporte le contenu stocké par le buffer.

e	5	12	10	9	8	7	1	3	...
w	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*	...
r	*false*	*false*	*true*	*false*	*true*	*false*	*true*	*true*	...
o	5	5	5	12	12	9	9	8	...

La figure 14 représente le contenu du buffer au cinquième instant, ainsi que les indices d'écriture et lecture. On a verdi les cases contenant des valeurs stockées depuis le début de l'exécution.

Question 20. *Écrivez une variante `ring_buffer_clocked` en utilisant les horloges pour spécifier que l'entrée `e` n'est présente que lorsque le flot `w` vaut vrai, tandis que la sortie `o` n'est présente que lorsque le flot `r` vaut vrai.*

Question 21 (Plus difficile)**.** *Écrivez une variante `ring_buffer_checked` dotée d'une sortie supplémentaire `err` qui est vraie lorsque `r` est vrai mais qu'il n'y a rien à lire, ou bien lorsque `w` est vrai mais qu'il n'y a plus d'espace pour écrire.*

Expressions et paramètres statiques. Dans l'exemple du buffer circulaire donné précédemment, la taille du tableau interne est dictée par la valeur d'une constante `n` déclarée au début du fichier. Avoir isolé cette constante de la sorte permet de changer très facilement la taille du tableau. Néanmoins, cette solution reste peu satisfaisante, puisqu'elle ne permet pas d'utiliser des buffers circulaires de tailles différentes dans un même programme, à moins de copier-coller le code.

Pour pallier ce défaut, Heptagon permet de déclarer des *paramètres statiques*, qui sont des paramètres inconnus mais dont la valeur doit être fixée à la compilation. Ils sont introduits entre double chevrons, avant les paramètres normaux du nœud, comme dans la variante paramétrée du buffer circulaire qui suit.

```
node ring_buffer<<n : int>>(e : int; w, r : bool)
                  returns (o : int)
var r_idx, w_idx : int; pa, a : int^n;
let
  o = a.[r_idx] default 0;
  pa = (0^n) fby a;
  a = if w then [ pa with [w_idx] = e ] else pa;
  r_idx = 0 fby (((if r then 1 else 0) + r_idx) % n);
```

```
  w_idx = 0 fby (((if w then 1 else 0) + w_idx) % n);
tel
```

Lors d'un appel, la liste des arguments statiques doit être fournie entre double chevrons également, avant les arguments normaux. Ainsi, le code ci-dessous appelle deux fois le buffer circulaire, chaque fois avec une taille différente.

```
node test_buffer() returns (r1 : bool; o1 : int;
                            r2 : bool; o2 : int)
var e : int; w : bool;
let
  e = 0 fby (e + 1);
  w = true fby true fby false fby false fby w;
  r1 = false fby false fby true fby true fby r1;
  o1 = ring_buffer<<2>>(e, w, r1);
  r2 = true fby false fby r2;
  o2 = ring_buffer<<1>>(e, w, r2);
tel
```

Question 22. *Donnez un chronogramme pour le nœud `test_buffer`. Que se passe-t-il si on passe l'argument statique `1` au buffer dont la sortie est `o1` ?*

Les expressions statiques sont celles qui apparaîssent dans les tailles des tableaux, dans les accès indicés statiques, ou encore comme arguments des nœuds disposant de paramètres statiques. Elles peuvent faire référence aux constantes globales (comme `n` dans la première implémentation du buffer circulaire), ou bien aux paramètres statiques du nœud courant (comme dans le code ci-dessus). Elles peuvent utiliser la plupart des opérateurs combinatoires d'Heptagon, notamment les opérateurs arithmétiques. Il est ainsi possible de déclarer une variable de type `int^(n + 1)` où `n` est une constante statique. En revanche, les opérateurs séquentiels comme `fby` ne sont pas autorisés.

Enfin, une dernière construction élémentaire de manipulation de tableaux est la concaténation `a1 @ a2`, qui résulte en un tableau dont la taille est la somme des tailles de `a1` et de `a2`.

Itérateurs. Les constructions sur les tableaux vues jusqu'à présent ne permettent que d'accéder à un nombre fixé de cases d'un tableau. Par exemple, nous ne pouvons pas exprimer un registre à décalage générique qui fonctionnerait pour toute taille de tableau. Heptagon propose sous le nom d'*itérateurs* des constructions pour parcourir les cases d'un tableau en effectuant divers traitement. Elles constituent des variantes des fonctions d'ordre supérieur classiques sur les séquences bien connues par les amateurs de langages fonctionnels. On va décrire les principaux itérateurs, en associant à chacun sa règle de typage.

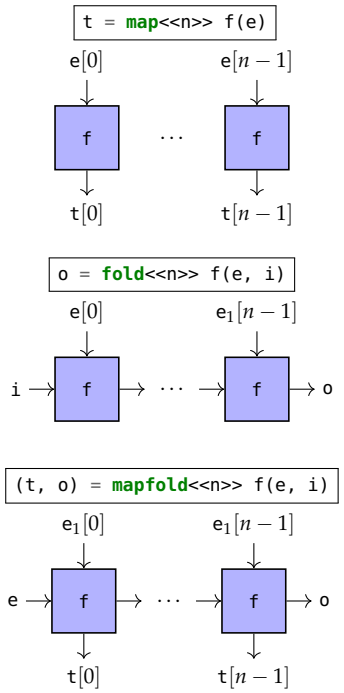


FIGURE 15: trois principaux itérateurs.

— La construction **map**<<n>> prend une fonction `f` recevant k entrées et produisant l sorties, et applique indépendamment `f` aux n éléments de k tableaux pour obtenir n éléments de l tableaux en sortie.

$$\frac{\mathtt{f} : \mathtt{a}_1 \times \cdots \times \mathtt{a}_k \to \mathtt{b}_1 \times \cdots \times \mathtt{b}_l \qquad \mathtt{e}_1 : \mathtt{a}_1\hat{}n \qquad \ldots \qquad \mathtt{e}_k : \mathtt{a}_k\hat{}n}{\mathbf{map}\texttt{<<n>>}\ \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_k) : \mathtt{b}_1\hat{}n \times \cdots \times \mathtt{b}_l\hat{}n}$$

— La construction **fold**<<n>> prend une fonction `f` recevant $k+1$ entrées et produisant une sortie, et l'applique successivement aux éléments de k tableaux de taille n.

$$\frac{\begin{array}{c}\mathtt{f} : \mathtt{a}_1 \times \cdots \times \mathtt{a}_k \times \mathtt{c} \to \mathtt{c} \\ \mathtt{e}_1 : \mathtt{a}_1\hat{}n \qquad \ldots \qquad \mathtt{e}_k : \mathtt{a}_k\hat{}n \qquad \mathtt{i} : \mathtt{c}\end{array}}{\mathbf{fold}\texttt{<<n>>}\ \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_k, \mathtt{i}) : \mathtt{c}}$$

— La construction **mapfold**<<n>> prend une fonction `f` recevant $k+1$ entrées et produisant $l+1$ sorties. Elle combine les deux constructions précédentes.

$$\frac{\begin{array}{c}\mathtt{f} : \mathtt{a}_1 \times \cdots \times \mathtt{a}_k \times \mathtt{c} \to \mathtt{b}_1 \times \cdots \times \mathtt{b}_l \times \mathtt{c} \\ \mathtt{e}_1 : \mathtt{a}_1\hat{}n \qquad \ldots \qquad \mathtt{e}_k : \mathtt{a}_k\hat{}n \qquad \mathtt{i} : \mathtt{c}\end{array}}{\mathbf{mapfold}\texttt{<<n>>}\ \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_k, \mathtt{i}) : \mathtt{b}_1\hat{}n \times \cdots \times \mathtt{b}_l\hat{}n \times \mathtt{c}}$$

La figure 15 représente graphiquement le fonctionnement de ces trois itérateurs. Pour éviter de surcharger visuellement cette figure, on a choisi de ne représenter que le cas où $k = l = 1$.

En plus des itérateurs **map**, **fold** et **mapfold**, Heptagon propose des variantes qui recoivent un argument supplémentaire correspondant à l'indice courant dans le tableau. Ces variantes, **mapi**, **foldi** et **mapfoldi**, peuvent être programmées en fonctions des précédentes mais sont si utiles qu'elles sont intégrées au langage. Nous vous renvoyons au manuel d'Heptagon pour plus de détails.

À l'aide des itérateurs, on peut finalement programmer un registre à décalage générique en la taille du tableau à mémoriser. Le code est donné ci-dessous. Il utilies une fonction auxiliaire `shiftr_aux` pour implémenter le décalage.

```
fun shiftr_aux(a, acc : bool) returns (b, newacc : bool)
let
  b = acc;
  newacc = a;
tel

node shiftr<<n : int>>(ini : bool^n; sh : bool)
               returns (o : bool)
var mem, nxt, sft : bool^n; d : bool;
```

```
let
  mem = ini fby nxt;
  (sft, d) = mapfold<<n>> shiftr_aux(mem, mem[n-1]);
  nxt = if sh then sft else mem;
  o = nxt[n-1];
tel
```

Question 23. *À quoi ressemble la représentation de* `mapfold` *donnée à la figure 15 lorsque vous remplacez* `f` *par le corps de* `shiftr_aux` *?*

Optimisations sur les tableaux. Les tableaux d'Heptagon sont *persistants*, c'est à dire qu'ils fabriquent de nouvelles valeurs sans modifier les tableaux existants. Cette propriété rend leur comportement facile à décrire mathématiquement ainsi qu'à comprendre. En revanche, cela signifie qu'une implémentation naïve est assez coûteuse. Par exemple, toute opération de modification de tableau `[ t with [i] = e ]` doit commencer par copier le tableau `t`, avant de modifier la case d'indice `i` de la copie. Le compilateur Heptagon emploie diverses optimisations pour minimiser le nombre de copies. Il essaie également de fusionner les itérateurs imbriqués de sorte à minimiser le nombre de parcours de chaque tableau et à éliminer les tableaux intermédiaires. Par exemple, lorsque `map<<n>> f` est appliqué à `map<<n>> g`, le compilateur construit une fonction `h` qui associe `f(g(x))` à `x`, et remplace les `map` imbriqués par `map<<n>> h`. Cette transformation, l'élimination de copie, et d'autres optimisations encore sont activées en passant l'option `-O` à `heptc`.

Pour en savoir plus, référez vous à l'article de Gérard, Guatto, Pasteur et Pouzet [7] au sujet de l'ajout de l'intégration des tableaux à un langage synchrone.

Applications

Programmation audio en temps-réel

La programmation audio en temps-réel produit et traite des flots d'échantillons sonores à une fréquence relativement élevé (44.1 kHz). Il se trouve que certains de ces traitements s'écrivent assez facilement dans un langage synchrone tel qu'Heptagon. Le sous-dossier `audio/` contient un exemple d'un tel programme écrit en Heptagon.

Contrôle d'un pendule inversé

Comme on l'a vu, la plupart des applications des langages synchrones vont chercher à contrôler des systèmes réactifs, en mettant en jeu des techniques issues de l'automatique. Un problème classique en automatique est de contrôler un pendule inversé, de sorte à maintenir son bras à la verticale au dessus d'un charriot. Le programme synchrone contenu dans le sous-dossier `inverted-pendulum/` simule ce problème, et expose deux techniques de contrôle : un contrôleur *bang-bang* et un contrôleur *proportionnel-intégral-dérivé*.

Compilation des langages synchrones à flots de données

Introduction

Le but de la dernière partie de ces notes est de vous fournir une introduction à la compilation des langages synchrones à flots de données tels que SCADE et Heptagon. Le développement de ces techniques a connu deux grandes périodes.

Compilation en automate. Des années 1980 jusqu'à la fin des années 1990, les compilateurs ont majoritairement reposé sur la théorie des automates. Il s'agit de construire explicitement l'automate fini qui est décrit par un programme synchrone. Cet automate peut ensuite être transformé en profondeur, par exemple subir une minimisation.

Compilation guidée par les horloges. Depuis le début des années 2000, la compilation des langages synchrones à flots de donnée s'est rapprochée de celle des langages plus classiques. Plutôt que de générer un automate explicite, on traduit progressivement le code source vers du code impératif qui implémente l'automate implicitement. Cette traduction exploite l'information d'horloge pour optimiser la structure de contrôle du code généré.

Ces deux familles d'approches partagent le même point d'arrivée : le compilateur synchrone produit un code source en C. Celui-ci peut ensuite être lui-même transformé en exécutable par la chaîne de compilation standard de la plateforme cible.

Dans cette partie du cours, nous allons discuter uniquement des techniques de compilation guidées par les horloges [21]. Si elles ont tendance à générer du code moins performant que celles de la première, elles sont capables de compiler chaque sous-programmes indépendamment, et peuvent donc traiter des programmes plus grands. De plus, elles s'étendent naturellement à des langages plus expressifs que Lustre. Ces raisons ont mené à l'adoption de la génération de code guidée par les horloges dans la plupart des compilateurs actuels, dont le compilateur industriel SCADE et le compilateur Heptagon.

Pour étudier les grandes étapes de la compilation des langages synchrones, nous allons nous intéresser à un compilateur existant. Le compilateur Heptagon est un choix naturel puisqu'il s'agit d'un logiciel libre dont le code source est librement consultable et modifiable. Comme tous les compilateurs modernes, il décompose la traduction d'un fichier source en fichier cible (ici, du code C) en plusieurs *passes de compilation*. De plus, il emploie un certain nombre de *langages intermédiaires*, c'est-à-dire de langages spécialisés qui facilitent l'implémentation de certaines analyses et transformation.

Une version simplifiée des différentes passes et langages intermédiaires du compilateur Heptagon est présentée à la figure 16. Les trois

21. Le lecteur intéressé par la compilation en automate pourra consulter l'article de Halbwachs et al. [8].

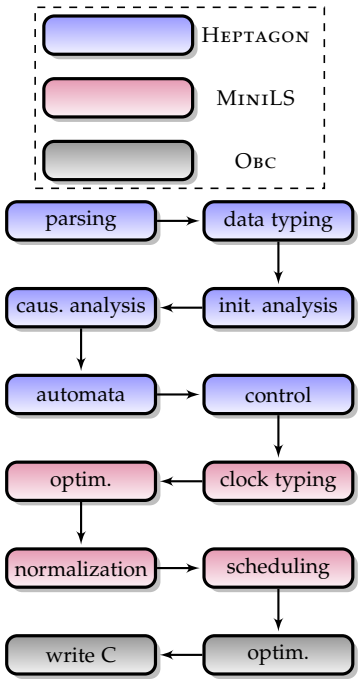


Figure 16: flot de compilation d'`heptc`.

langages connus du compilateur sont le langage source (*Heptagon*), un langage intermédiaire sans structure de contrôle (*MiniLS*) et un langage intermédiaire impératif (*Obc*). Le processus de compilation se découpe en trois phases schématiques.

1. Le code source Heptagon est soumis à plusieurs analyses statiques légères qui assurent l'absence de certaines erreurs à l'exécutions (*data typing*, *causality analysis* et *initialization analysis*).
2. Il est ensuite transformé de sorte à éliminer les structures de contrôle, les automates au premier titre (*automata*, *control*). Le code qui en résulte est traduit en MiniLS, un langage purement équationnel, sans aucune construction de contrôle.
3. Le compilateur calcul un ordre total entre les équations qui forment le code MiniLS. Cet ordre doit respecter les dépendances et permet de voir chaque équation comme une instruction qui modifie l'état courant (*scheduling*). Les équations peuvent ensuite être traduites vers du code impératif implémentant une fonction de transition.

En plus de ces étapes essentielles, le compilateur Heptagon applique un certain nombre d'optimisations pour améliorer la taille du code généré, sa performance ou l'espace requis (*optim*). Toutes ces étapes de traduction sont appliquées à chaque nœud indépendamment.

Le reste de cette section sera consacrée à décrire chaque étape du processus de traduction. Plutôt que de partir du langage source pour arriver au code final, nous allons plutôt procéder en arrière : on va d'abord expliquer la traduction de MiniLS vers du code impératif, avant de détailler l'élimination des structures de contrôle d'Heptagon, et finir avec les analyses statiques appliquées au code source. En d'autres termes, on va parcourir la figure 16 de bas en haut.

Plusieurs constructions et traits d'Heptagon que nous avons utilisés dans les sections précédentes ne vont pas être traités. Par exemple, nous ne discuterons pas des constructions liées aux tableaux, y compris les itérateurs. Nous ne traiterons pas non plus des paramètres statiques. Enfin, nous ne couvrirons pas les optimisations utilisées par `heptc`, bien que certaines d'entre elles soient assez importantes en pratique — notamment l'élimination des copies de tableaux superflues.

De MiniLS à Obc

Mémoires. Le langage intermédiaire MiniLS peut être vu comme un fragment d'Heptagon. Ce fragment est purement équationnel, au sens où on n'y trouve aucune construction de contrôle de haut niveau : pas d'automates, de **`switch`**, de **`present`**, ou de **`if`** sur les équations. Les variables **`last`** sont également absentes. En revanche, il dispose toujours de **`when`** et **`merge`**, ainsi que de **`fby`**. La construction **`reset`** est toujours

présente, mais elle ne peut s'appliquer qu'à une application de fonction, c'est-à-dire qu'elle est restreinte à la forme **reset** $f(e_1, \ldots, e_n)$ **every** x.

Comme premier exemple de l'effet du processus de compilation de MiniLS, revenons à un de nos premiers exemples, celui du flot des entiers naturels. Il se trouve être écrit directement dans le fragment d'Heptagon qui correspond à MiniLS, c'est à dire qu'il ne contient pas de structure de contrôle.

```
(* Code source en Heptagon/MiniLS. *)
node c0() returns (nat : int)
let
  nat = 0 fby (nat + 1);
tel
```

Les passes de compilation qui traitent les structures de contrôle (en bleu à la figure 16) sont ici sans effet. Intéressons nous plutôt aux passes en rouge. Le calcul d'horloges est également sans effet, puisque ce nœud ne contient ni `when`, ni `merge`, et n'appelle aucun nœud qui contiendrait ces constructions. La première passe intéressante est la normalisation. Celle-ci fait en sorte que certaines constructions, notamment **fby**, soient dans leur propre équation, en leur attribuant un nom (ici `mem_nat`).

```
(* Code MiniLS normalisé. *)
node c0() returns (nat : int)
var m_nat : int; v : int;
let
  v = (nat + 1);
  m_nat = 0 fby v;
  nat = m_nat
tel
```

Une fois la normalisation effectuée, le code impératif est à portée de vue. En effet, chaque nom assigné à une sous-expression **fby** va correspondre à une *variable d'état* qui devra persister entre chaque réaction, à l'inverse des autres variables, purement locales. Toutefois, il reste à décider de l'ordre dans lequel les calculs doivent être effectués. C'est le rôle de la passe d'*ordonnancement*, qui produit le code suivant.

```
(* Code MiniLS ordonnancé. *)
node c0() returns (nat : int)
var m_nat : int; v : int;
let
  nat = m_nat;
  v = (nat + 1);
  m_nat = 0 fby v;
tel
```

On voit que le code ordonnancé est une permutation du code normalisé. Formellement, la normalisation calcule un ordre strict $<$ entre équations tel que, pour toute paire d'équations (E_1, E_2),

1. si E_1 n'est pas un **fby** et dépend instantanément d'une valeur calculée par E_2, et que E_2 n'est pas un **fby**, alors $E_2 < E_1$,
2. si E_1 est de la forme $x = c$ **fby** e et e dépend d'une valeur calculée par E_2, et que E_2 n'est pas un **fby**, alors $E_2 < E_1$,
3. si E_1 est de la forme $x = c$ **fby** e et E_2 dépend instantanément de x, alors $E_2 < E_1$.

La troisième propriété assure que toutes les lectures d'une variable définissant un délai sont effectuées après sa définition. Cette dernière peut donc être effectuée en place, de façon impérative. C'est l'idée derrière le langage intermédiaire Obc, qui est bien un langage impératif où chaque nœud `c0` se voit traduit en un "objet" [22] modifiable appelé *machine*.

22. Si la terminologie vient de la programmation orientée objet, celle-ci reste techniquement assez lointaine : pas d'héritage ou de liaison tardive ici.

```
-- Code impératif dans le langage Obc.
machine c0 =
  var m_nat: int;

  reset() returns () {
    mem(m_nat) = 0
  }

  step() returns (nat: int) {
    var v: int;
    nat = mem(m_nat);
    v = ((+) nat 1);
    mem(m_nat) = v
  }
```

Un programme Obc est donc constitué par un ensemble de machines. Chaque machine spécifie un ensemble de *mémoires*, une méthode `step` et une méthode `reset`. Les mémoires correspondent aux équations MiniLS qui définissaient des délais. Le corps de la méthode `reset` réinitialise les variables d'instances, en y écrivant les valeurs initiales des délais. Le corps de la méthode `step` calcule les sorties en fonction des entrées (`c0` n'en a pas) et de la valeur courante des mémoires, et met à jour celles-ci. Ce corps est formé d'une séquence d'instructions qui correspondent directement aux équations du code MiniLS ordonnancé.

Obc étant un langage impératif très simple, il peut facilement être traduit vers toutes sortes de langages : C, Javascript, OCaml, Ada, etc. Le compilateur Heptagon effectue encore quelques optimisations sur Obc. Une fois ceci-fait, il ne reste essentiellement qu'à afficher les constructions d'Obc dans la syntaxe du langage cible. À titre d'exemple,

le code C ci-dessous est le produit final de la compilation. Les mémoires de la machine ont été placées dans une structure dédiée, tout comme ses sorties.

```
/* Code C final. */
typedef struct c0_mem {
  int m_nat;
} c0_mem;

typedef struct c0_out {
  int nat;
} c0_out;

void c0_reset(c0_mem *self) {
  self->m_nat = 0;
}

void c0_step(c0_out *_out, c0_mem *self) {
  int v;
  _out->nat = self->m_nat;
  v = (_out->nat+1);
  self->m_nat = v;;
}
```

On peut ensuite produire du code exécutable avec n'importe quel compilateur C, et lancer le programme en appelant la fonction `c0_step()` de façon répétée, par exemple à intervalle de temps physique fixe.

Instances. On a vu que tout usage d'un délai donnait lieu à l'usage d'une mémoire dans le code final. Lorsqu'un nœud `f` applique un nœud `g`, il faut donc qu'une nouvelle copie des mémoires de `g` soit créée. Considérons le code ci-dessous.

```
(* Code source en Heptagon/MiniLS. *)
node c1() returns (o : int)
let
  o = c0() + c0() + (0 fby o);
tel
```

Une fois ce code MiniLS normalisé et ordonnancé, il est traduit par le compilateur Heptagon vers la machine Obc suivante.

```
machine c1 =
  var v_3: int;
  obj c0_1 : c0; c0 : c0;

  reset() returns () {
```

```
    c0_1.reset();
    c0.reset();
    mem(v_3) = 0
  }

  step() returns (o: int) {
    var v_2: int; v_1: int; v: int;
    (v_1) = c0_1.step();
    (v) = c0.step();
    v_2 = ((+) v v_1);
    o = ((+) v_2 mem(v_3));
    mem(v_3) = o
  }
```

En plus de la mémoire `v_3`, la machine `c1` contient deux *instances*, c'est-à-dire des sous-machines nommées. Chacune des deux instances correspond à un appel à `c0` depuis `c1`, et stocke les mémoires correspondantent. Observons que la méthode `reset` de la machine `c1` réinitialise récursivement ces deux instances, pour que leurs mémoires soient également réinitialisées. La méthode `step` fait appel aux méthodes `step` des instances pour calculer leurs sorties respectives.

Réinitialisation. La mémoire du code généré par Heptagon est donc arborescente : chaque nœud donne lieu à une machine dont la mémoire contient une copie de la mémoire de chacune de ses sous-machines, et ce récursivement. Chaque sous-machine correspondant à un appel de nœud dans le code source, éviter ces appels peut parfois être judicieux. Par exemple dans le cas du nœud `c1` précédent, une seule copie de `c0` aurait suffit, et on aurait pu multiplier sa sortie par deux. Dans la plupart des cas, c'est impossible. Considérons par exemple le nœud `c2` suivant.

```
(* Code source en Heptagon/MiniLS. *)
node c2() returns (o : int)
var h : bool; a, b : int;
let
  h = true fby not h;
  a = c0();
  reset b = c0() every h;
  o = a + b;
tel
```

Il est compilé vers la machine Obc ci-dessous.

```
machine c2 =
  var h: bool;
```

```
obj c0_1 : c0; c0 : c0;

reset() returns () {
  c0_1.reset();
  c0.reset();
  mem(h) = true
}

step() returns (o: int) {
  var v: bool; a: int; b: int;
  switch (mem(h)) {
    case true:
      c0_1.reset()
  };
  (b) = c0_1.step();
  (a) = c0.step();
  o = ((+) a b);
  v = not(mem(h));
  mem(h) = v
}
```

La construction **reset/every** a été traduite vers un appel à la méthode **reset** de l'instance `c0_1` lorsque la mémoire `h` contient le booléen vrai à la réaction courante. On voit ici que les deux instances ne peuvent pas être fusionnées, puisqu'elles transportent des valeurs distinctes.

Contrôle et horloges. Seules deux constructions de MiniLS peuvent induire la présence de conditionnelles dans le code généré ;

1. la réinitialisation (cf. exemple précédent),
2. les horloges.

Pour voir comment les horloges interagissent avec le processus de génération de code, considérons l'exemple suivant.

```
(* Code source en Heptagon/MiniLS. *)
node c3(c : bool) returns (o : int)
var a, b : int;
let
  a = c0 ();
  b = c0 ();
  o = merge c a (b whenot c);
tel
```

Question 24. *Donnez un chronogramme montrant les valeurs des variables* `a`, `b` *et* `o` *pour* `c` *valant true* :: *false* :: *false* :: *true* :: . . . *Déduisez-en les horloges de ces trois variables.*

En écrivant un chronogramme, on réalise que `b` et `o` sont sur l'horloge de base "`.`", tandis que `a` est sur l'horloge "`. on c`". Ces horloges dictent les réactions auxquelles les instances correspondantes doivent être calculées dans la méthode `step` générée.

```
machine c2 =
  obj c0_1 : c0;c0 : c0;

  reset() returns () {
    c0_1.reset();
    c0.reset()
  }

  step(c: bool) returns (o: int) {
    var a: int; b: int;
    (b) = c0_1.step();
    switch (c) {
      case true:
        (a) = c0.step();
        o = a
      case false:
        o = b
    }
  }
```

Question 25. *Que se passe-t-il si l'appel à `c0_1.step()` est déplacé dans la deuxième branche de la conditionnelle ?*

Récapitulatif La traduction de MiniLS vers du code impératif obéit donc aux principes suivants.

1. Un nœud est traduit vers une machine, qui réunit un ensemble de mémoires, d'instances et de méthodes.
   — Chaque délai donne lieu à une mémoire,
   — chaque appel de nœud donne lieu à une instance.
2. Les équations MiniLS sont traduites vers des instructions qui constituent le corps de la méthode `step` de la machine.
   — Elles doivent être ordonnées pour respecter les dépendances…
   — et permettre de mettre à jour en place les mémoires.
3. La réinitialisation est implémentée en appelant la méthode `reset` de l'instance correspondante.
4. Les horloges déterminent à quelles réactions chaque instruction doit être calculée. Elles sont traduites vers des conditions booléennes qui gardent l'exécution des instructions.

De Heptagon à MiniLS

Le compilateur traduit progressivement les structures de contrôle de haut niveau en équations simples. Ce processus est structuré en deux grandes étapes : d'abord l'élimination des automates, puis l'élimination des autres structures de contrôle. Le code résultat est finalement traduit en MiniLS. Cette sous-section détaille ces traductions par l'exemple, sans donner leur forme générale, qui peut être trouvée dans les articles scientifiques idoines.

Élimination des automates. Pour comprendre comment le compilateur Heptagon éliminer les automates hiérarchiques, on va observer son action en commençant par du code très simple.

La conception générale des automates ainsi que le schéma général de leur traduction telle qu'implémentée dans le compilateur Heptagon est présenté par les articles suivants de Colaço, Hamon, Pagano et Pouzet.

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; and J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006

```
node f(x : bool) returns (o : int)
let
  automaton
    state A
      do o = 0 fby (o + 1)
      unless x continue B

    state B
      do o = 1 fby (2 * o)
      until x continue A
  end
tel
```

Après élimination des automates, on obtient le code suivant.

```
type st = St_A | St_B
node f(x : bool) returns (o : int)
var s, ns : st; r, nr, pnr : bool;
let
  switch (St_A fby ns)
    | St_A
      do reset (s, r) =
           if x then (St_B, false) else (St_A, pnr)
         every pnr
    | St_B
      do reset (s, r) = (St_B, pnr)
         every pnr
  end;
  switch (s)
    | St_A
      do reset (ns, nr) = (St_A, false);
                      o = 0 fby (o + 1)
```

```
        every r
    | St_B
      do reset (ns, nr) = if x then (St_A, false)
                               else (St_B, false);
                     o = 1 fby (2 * o)
        every r
  end;
  pnr = false fby nr
tel
```

Le nœud obtenu est le fruit d'une transformation générale et systématique réduisant les automates hiérarchiques à la construction `switch`. Pour cette raison, il contient une certaine quantité de redondance.

Question 26. *Comment évoluent les flots `r`, `nr` et `pnr` au cours du temps ?*

Un peu de réflexion permet de se convaincre que ces trois flots valent constamment `false`. On peut donc simplifier manuellement le code obtenu pour obtenir un équivalent plus simple et facile à lire.

```
node f(x : bool) returns (o : int)
var s : st; ns : st;
let
  switch St_A fby ns
    | St_A do s = if x then St_B else St_A
    | St_B do s = St_B
  end;
  switch s
    | St_A do ns = St_A; o = 0 fby (o + 1)
    | St_B do ns = if x then St_A else St_B;
              o = 1 fby (2 * o)
  end;
tel
```

Comment lier ce code au code à l'automate de départ ? Tout d'abord, on peut constater que le compilateur a défini un nouveau type `st` dont les valeurs sont les états potentiels de l'automate, ici nommés `St_A` et `St_B`. Ensuite, que la passe d'élimination des automates a également introduit deux nouvelles variables locales de type `st`, à savoir `s` et `ns`. Le flot `s` transporte l'état courant de l'automate, tandis que `ns` transporte l'état qui sera le sien au prochain instant — sauf si une transition `unless` se produit. La présence de deux utilisations du mot-clef `switch` reflète cette distinction entre état courant et prochain état.

— Le premier bloc `switch` calcule l'état courant en fonction de la valeur précédente de `ns` et de l'entrée `x`. Au premier instant, tout se passe comme si l'automate avait été démarré dans l'état initial `A`.

Si l'on a déterminé à l'instant précédent que le nouvel état devrait être `St_B`, alors l'état courant est toujours `St_B`, reflétant l'absence de transition forte dans l'état `B`. En revanche, si l'on a déterminé à l'instant précédent que le nouvel état devrait être `St_A`, l'équation `s = if x then St_B else St_A` fixe l'état courant en fonction de `x`. Ce comportement reflète la présence d'une transition forte sur `x` dans l'état `A`.

— Le second bloc **`switch`** calcule le prochain état prévu ainsi que la valeur de la sortie `o` en fonction de l'état courant. Le calcul du prochain état reflète la présence ou l'absence de transitions faibles à l'état courant. Ainsi, aucune transition faible n'est présente à l'état `A`, ce qui est reflété par l'équation `ns = St_A`. À l'inverse, l'équation `ns = if x then St_A else St_B` traduit la transition **`until`** `x` **`then`** `A` dans l'état `B`.

La traduction générale des automates suit le schéma que nous venons d'illustrer sur un exemple. Le point le plus important est l'utilisation de deux flots distincts pour représenter l'état courant de l'automate, et l'état de l'automate à l'instant suivant *en l'absence de transition forte*.

Notre exemple précédent ne disposait que de transitions continuantes. Pour comprendre la traduction des transitions réinitialisantes, on peut observer le résultat de la traduction une fois le mot-clef **`continue`** remplacé par **`then`**. On obtient alors le code ci-dessous.

```
type st = St_A | St_B
node f(x : bool) returns (o : int)
var s, ns : st; r, nr, pnr : bool;
let
  switch St_A fby ns
    | St_A do reset (s, r) = if x then (St_B, true)
                                  else (St_A, pnr)
             every pnr
    | St_B
      do reset (s, r) = (St_B, pnr) every pnr
  end;
  switch s
    | St_A
      do reset (ns, nr) = (St_A, false);
                      o = 0 fby (o + 1)
         every r
    | St_B
      reset
        (ns, nr) = if x then (St_A, true) else (St_B, false);
               o = 1 fby 2 * o
      every r
```

```
  end;
  pnr = false fby nr
tel
```

Cette fois-ci, les flots `r`, `nr` et `pnr` jouent un rôle non trivial. Le flot `r` est vrai lorsqu'il faut réinitialiser l'état courant de l'automate. Le flot `nr` est vrai lorsqu'il faudra réinitialiser l'état de l'automate à l'instant suivant. Le flot `pnr` est vrai lorsqu'on a décidé à l'instant précédent qu'il faudrait réinitialiser l'état de l'automate pour l'instant courant, sauf si une transition forte est prise. En particulier, le flot `r` contrôle la réinitialisation (via le mot-clef **`reset`**) des équations présent dans le corps d'un état de l'automate.

Enfin, remarquons que le code généré reflète une spécificité de la sémantique des automates d'Heptagon. Il s'agit de la possibilité, expliqué à la section , pour un état d'être franchi instantanément. Cette situation ne se produit que lorsque qu'on y est entré via une transition faible à la fin de l'instant précédent pour en sortir immédiatement au début de l'instant courant. Toutefois, si la transition faible est réinitialisante, l'état intermédiaire doit tout de même être réinitialisé !

Élimination du contrôle. Le code produit après élimination des automates contient toujours des constructions de contrôle qui permettent d'activer des blocs d'équations sporadiquement, et autorisent la définition d'une variable dans plusieurs blocs. On va se focaliser sur la construction **`switch`**, le traitement de **`if`** et de **`present`** étant similaire.

Revenons au code obtenu après élimination du tout premier automate. En continuant le processus de compilation, on obtient le résultat ci-dessous.

```
node f(x : bool) returns (o : int)
var s, ns : st; s_St_A, s_St_B, ck : st;
    ns_St_A, ns_St_B, ck_1 : st; o_St_A, o_St_B : int;
let
    (* Traduction du premier switch. *)
    s = merge ck (St_B -> s_St_B)(St_A -> s_St_A);
      (* Première branche du premier switch. *)
      s_St_B = St_B;
      (* Seconde branche du premier switch. *)
      s_St_A = if (x when St_A(ck)) then St_B else St_A
    ck = St_A fby ns;
    (* Traduction du second switch. *)
    ns = merge ck_1 (St_B -> ns_St_B)(St_A -> ns_St_A);
    o = merge ck_1 (St_B -> o_St_B)(St_A -> o_St_A);
      (* Première branche du second switch. *)
      ns_St_B = if (x when St_B(ck_1)) then St_A else St_B;
```

```
    o_St_B = 1 fby (2 * (o when St_B(ck_1)));
    (* Seconde branche du second switch. *)
    ns_St_A = St_A;
    o_St_A = 0 fby ((o when St_A(ck_1)) + 1);
  ck_1 = s
tel
```

La traduction réalisée est plus simple qui élimine les automates. Elle s'appuie sur les opérateurs de sélection et de fusion, ainsi que sur la notion d'horloge. La condition qui détermine la branche active de chaque **switch** donne lieu à une nouvelle variable d'horloge, ici `ck` pour le premier **switch** et `ck1` pour le second. L'essentiel de la traduction consiste à distinguer soigneusement les définitions et usages d'une variable `s` déclarée à l'extérieur du **switch** – les variables déclarées localement peuvent être traduites telles quelles. Ainsi, les deux définitions de la variable `s` donnent lieu à deux variables distinctes `s_St_A` et `s_St_B`, qui sont fusionnées selon `ck`. Ces variables étant d'horloges lentes, elles doivent appliquer l'opérateur de sélection aux variables définies à l'extérieur du **switch**, par exemple `x` dans la définition de `s_St_A`.

Élimination de la réinitialisation par bloc. La construction de réinitialisation de MiniLS ne s'applique qu'aux appels de fonctions. Il faut donc ramener la construction générale **reset** `bloc` **every** `c` d'Heptagon à ce cas particulier. Ce processus consiste essentiellement à introduire des conditionnelles autour des mémoires (**fby**, **pre** et **->**).

Élimination des mémoires partagées. Enfin, MiniLS ne dispose pas de mémorie partagées (variables **last**), il faut donc éliminer celles-ci. Pour comprendre la traduction, considérons un exemple très simple de programme utilisant **last**.

```
node f(x : int) returns (last o : int = 0)
let
  o = x + last o;
tel
```

Le code MiniLS obtenu est le suivant.

```
node f(x : int) returns (o : int)
var o_1 : int;
let
  o_1 = 0 fby o;
  o = (x + o_1)
tel
```

La traduction consiste simplement à introduire une nouvelle variable qui représente la valeur précédente de `o`, ici `o_1`. Cette variable est le résultat d'un opérateur **`fby`** initialisé avec la valeur spécifiée lors de la déclaration de `o` comme mémoire. En l'absence de valeur d'initialisation déclarée, la définition de `o_1` utilisera plutôt l'opérateur **`pre`**.

L'élimination des **`last`** est réalisée en plusieurs étapes. Une partie est effectuée lors de l'élimination des automates. L'autre lors d'une passe idoine réalisée juste avant la production de code MiniLS, et qui n'a donc pas à se soucier des structures de contrôle.

Types, initialisation, causalité, horloges

Les programmes Heptagon sont soumis à une batterie d'analyses qui visent à interdir toute une classe d'erreurs. Ces analyses sont *statiques* : elles n'ont pas besoin d'exécuter le programme, et donc de disposer des données d'entrées. En contrepartie, elles sont imprécises, en ce qu'elles vont avoir tendance à rejeter des programmes corrects. Ce compromis familier est au cœur de la famille d'analyses statiques la plus populaire : les systèmes de types. Par exemple, une expression C comme `(x && false ? NULL : 42.f)` s'évalue toujours vers le nombre à virgule flottante `42.f` mais est rejetée comme incorrecte.

Dans le cas d'Heptagon, les analyses statiques sont au nombre de quatre : typage de données, analyse d'initialisation, analyse de causalité, calcul d'horloge. Le typage de données, l'analyse d'initialisation et le calcul d'horloge sont des systèmes de types. Les systèmes de types, en règle général, ont l'avantage d'être modulaires : ils permettent de traiter un nœud `f` comme une boîte noire, à partir du moment où le type de `f` est connu. Les quatre analyses sont appliquées au code source Heptagon, à l'exception du calcul d'horloge qui est appliqué sur MiniLS, et donc après l'élimination des structures de contrôle de haut niveau. On décrit brièvement chacune d'entre elles.

Types de données. Le système de types de données n'a rien de remarquable, et se rapproche de celui d'un langage comme Pascal. La grammaire qui décrit un type de données τ_d est

$$\tau_d ::= s \mid \tau_d \times \cdots \times \tau_d \mid \tau_d\text{^n}$$

où s est un nom de type définir, qu'il s'agisse d'un type de base prédéclaré comme **`int`** ou **`float`**, ou d'un type énuméré ou enregistrement défini par l'utilisateur. L'absence de type fonctionnel trahit le fait qu'un nœud Heptagon ne peut ni renvoyer ni recevoir en paramètre un autre nœud : le langage est *de premier ordre*. Les types sont vérifiés à partir des déclarations de variables, et non inférés comme dans un langage comme OCaml.

Initialisation. L'analyse d'initialisation vise à assurer que la valeur `nil` produite par chaque utilisation de l'opérateur **`pre`** au premier instant n'ait pas d'impact sur le comportement final du programme. Elle est implémentée comme un système de types. La grammaire qui décrit un type d'initialisation τ_i est

$$\tau_i ::= \mathbf{0} \mid \mathbf{1} \mid \tau_i \times \cdots \times \tau_i.$$

où les deux seuls types de base $\mathbf{0}$, qui classifie les valeurs *toujours initialisées*, et $\mathbf{1}$, qui classifie les valeurs *potentiellement non-initialisées*. Ces deux types sont liés par une relation de *sous-typage* qui spécifie $\mathbf{0} \leq \mathbf{1}$. Celle-ci reflète qu'il n'est pas incorrect d'oublier qu'une valeur est initialisée pour prétendre qu'elle ne l'est potentiellement pas. Elle permet d'utiliser une expression initialisée partout où une expression potentiellement non-initialisée peut convenir.

Comme tout système de types, l'analyse d'initialisation spécifie pour chaque opération du langage une règle de déduction qui permet de déduire du type des arguments le type du résultat. Les règles les plus intéressantes sont les suivantes.

On présente les règles de déduction dans un style informel mais intuitif. Le lecteur intéressé pourra se référer à l'article de Jean-Louis Colaço et Marc Pouzet [6], qui sert de base àa l'implémentation réalisée dans le compilateur Heptagon.

$$\textsc{Const}\ \frac{}{c : \mathbf{0}} \qquad \textsc{Op}\ \frac{\texttt{e1} : \tau_i^1 \qquad \texttt{e2} : \tau_i^2}{\texttt{op(e1, e2)} : \max(\tau_i^1, \tau_i^2)} \qquad \textsc{Pre}\ \frac{}{\texttt{pre x} : \mathbf{1}}$$

$$\textsc{Fby}\ \frac{\texttt{e1} : \tau_i^1 \qquad \texttt{e2} : \tau_i^2}{\texttt{e1 fby e2} : \tau_i^1} \qquad \textsc{LastInit}\ \frac{\texttt{last x : ... = v}}{\texttt{last x} : \mathbf{0}} \qquad \textsc{LastNoninit}\ \frac{\texttt{last x : ... = v}}{\texttt{last x} : \mathbf{1}}$$

Les constantes sont toujours initialisées. Le résultat d'un opérateur comme l'addition est potentiellement non initialisé dès que l'un de ses arguments l'est. Le résultat d'un **`pre`** n'est par définition pas initialisé. Le résultat de `e1 fby e2` est aussi initialisé que l'est `e1`. Enfin, une variable **`last`** est initialisée si sa déclaration fourni une valeur d'initialisation.

Causalité. L'analyse de causalité actuellement implémentée dans Heptagon ne prend pas la forme d'un système de types, mais d'une analyse à base de contraintes très simples. Elle est toutefois peu modulaire, puisqu'elle suppose que tous les résultats d'un appel de nœud dépendent de tous les arguments.

Une *contrainte de causalité* peut être vue comme une formule logique qui décrit un ordre entre les équations du nœud dont on cherche à vérifier la causalité. La grammaire qui décrit les contraintes C est

$$C ::= \texttt{write}(x) \mid \texttt{read}(x) \mid \top \mid C \wedge C \mid C < C \mid (C, \ldots, C)$$

où x dénote une variable du nœud en considération. Chaque nœud

La syntaxe réelle utilisée par Heptagon inclut la disjonction de contrainte $C \vee C$. Elle est traitée séparément des autres par une mise en forme normale disjonctive initiale. On néglige ce cas pour ne pas alourdir la présentation.

se voit associer une contrainte par l'analyse de causalité, et est dit *causal* si cette contrainte est résoluble. Intuitivement, une contrainte est résoluble si on peut construire une fonction σ qui associe à chaque contrainte C un entier n d'une façon "compatible" avec C. Par exemple, si $\texttt{write}(x)$ et $\texttt{write}(x)$ apparaissent tous deux dans C, alors on doit avoir $\sigma(\texttt{write}(x)) < \sigma(\texttt{read}(x))$, indiquant qu'une variable doit être écrite avant d'être lue. Pour vérifier qu'il existe un tel σ, Heptagon traduit la contrainte vers un graphe fini dont l'acyclicité équivaut à l'existence d'une solution.

Calcul d'horloge. Le calcul d'horloge est un système de types dont nous avons vu les grandes lignes lors de l'introduction des opérateurs de sélection et fusion (). Il est implémenté après l'élimination des structures de contrôle, dont les automates. C'est un défaut de la conception actuelle du compilateur, dans la mesure où un message d'erreur risque d'exposer l'utilisateur au code intermédiaire produit après l'élimination des structures de contrôle.

Références

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 2003.

[2] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *Principles of Programming Languages (POPL'13)*. Association for Computing Machinery, 2013.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL'87)*. Association for Computing Machinery, 1987.

[4] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[5] 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.

[6] J.-L. Colaço and M. Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.

[7] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A Modular Memory Optimization for Synchronous Data-flow Languages : Application to Arrays in a Lustre Compiler. In *Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'12)*. ACM, 2012.

[8] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing Congress (IFIP'74)*. IFIP, 1974.

[10] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9) :1321–1336, 1991.

[11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer New York, 1992.

[12] K. J. Åström and R. M. Murray. *Feedback Systems : An Introduction for Scientists and Engineers*. Princeton University Press, Jan 2008.