# *Programmation synchrone*

*Adrien Guatto*

*2020–2021*

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 l'année universitaire 2020–2021 à l'Université de Paris, en Master 2 Informatique ainsi qu'à l'École d'Ingénieur Denis Diderot.

Version du 13 octobre 2020. Je suis preneur de toute coquille, erreur ou remarque par courriel à l'adresse adrien@guatto.org.



## *Table des matières*

## *Introduction*

(Début du cours 01).

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 [6].

### *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 ?*

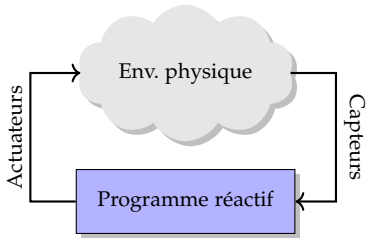



FIGURE 1: système réactif générique.

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.

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

*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.

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].

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.

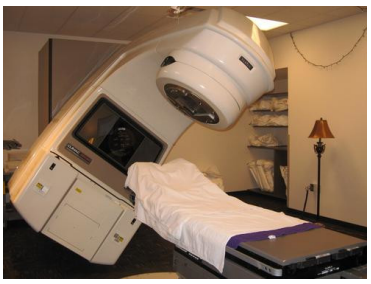


FIGURE 3: machine Therac-25.

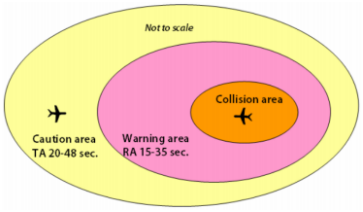



FIGURE 4: Zones d'intérêt du système d'évitement de collision TCAS II.

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].

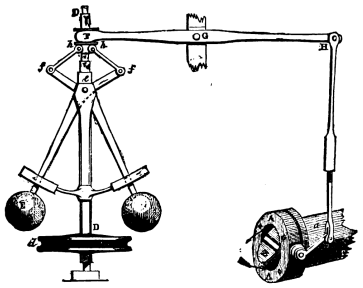
*Contraintes de ressources.* Enfin, les programmes réactifs se retrouvent en général exécutés par du matériel informatique dont c'est la seule fonction. On parle typiquement de *système embarqué*. De tels systèmes sont généralement soumis à des impératifs de coûts forts, surtout lorsqu'ils doivent être 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 les aspects 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 [7] 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.

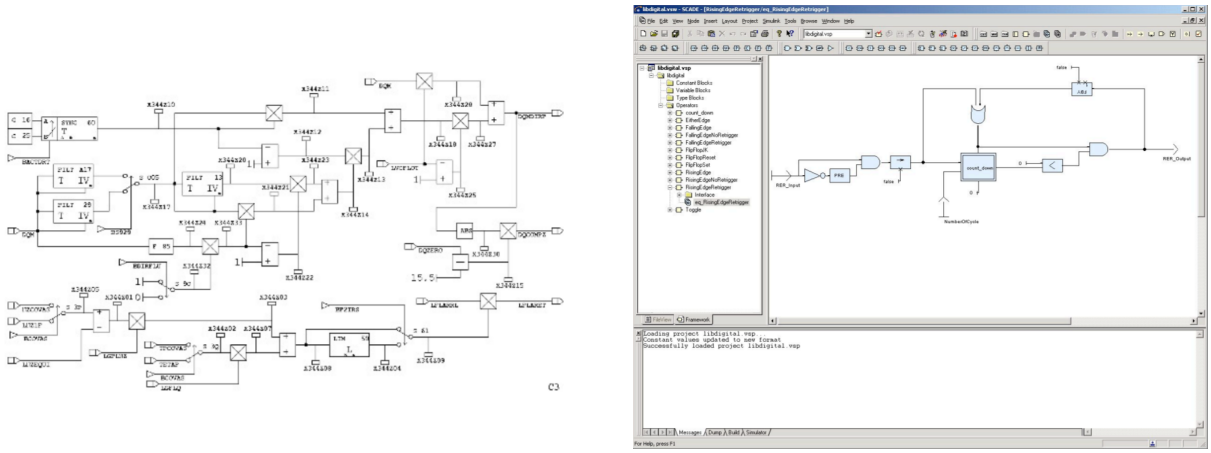


FIGURE 5: les langages synchrones à flots de données, du protolangage Spécification Assistée par Ordinateur (SAO) d'Airbus (à gauche), au langage industriel contemporain SCADE 6, développé par la compagnie Ansys (à droite).

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.

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 approche expérimentale de la conception de contrôleurs.

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 initiation à la sémantique formelle des langages synchrones sera indispensable.

## *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* :

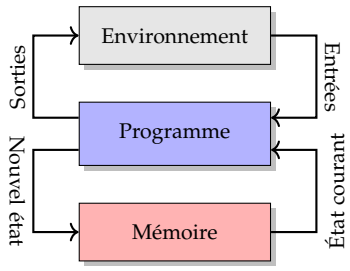



FIGURE 6: système réactif cyclique.

*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.

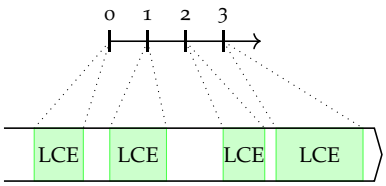



FIGURE 7: temps logique, temps physique.

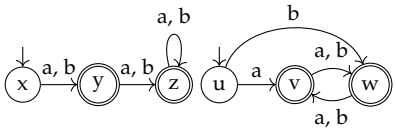
Le choix de structurer l'exécution comme une suite infinie d'interactions introduit naturellement une notion de *temps logique*. Ce temps logique est constituée d'une succession d'*instants logiques*, chacun d'eux correspondant à une interaction entre programme et environnement. Le caractère *logique* 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 $\Sigma_1$ est l'alphabet du mot d'entrée et $\Sigma_2$ 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 $\Sigma_1^*$ et $\Sigma_2^*$, mais des suites infinies de lettres, aussi appelée *flot*, et dont les ensembles sont dénotés $\Sigma_1^\omega$ et $\Sigma_2^\omega$. 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].

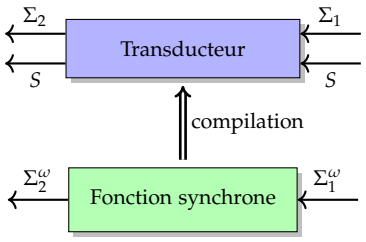



FIGURE 9: compilation des langages synchrones à flots de données.

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 [4].

### *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*

(Début du cours 02).

*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 `appel_addition`, 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 `appel_addition` 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.constantes`. 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 `x` 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

$$[\![e_1 \ \mathtt{fby}\ e_2]\!] = [\![e_1]\!]_1 :: [\![e_2]\!].$$

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 $[\![\texttt{pre}\ e]\!] = nil :: [\![e]\!]$.

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

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 sui.

```
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 première 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`).

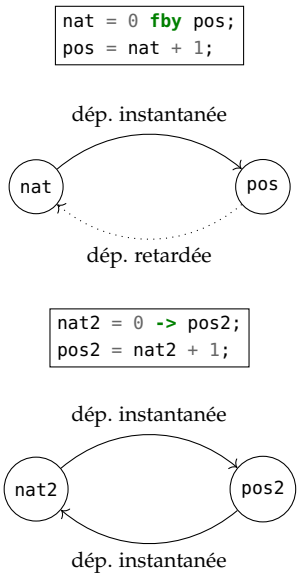



FIGURE 10: dépendances et causalité.

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. [5] 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.

(Début du cours 03.)

*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* | ... |

Il est en général recommandé d'éviter d'utiliser directement la réinitialisation modulaire. On verra que **`reset`** `block` **`every`** `c` est surtout en interne par les 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

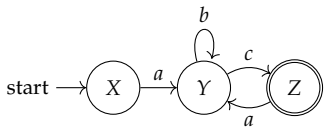



FIGURE 11: automate pour $(ab^*c)^+$.

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 on traite de fonctions synchrones. Néanmoins, les langages synchrones comme Heptagon ou SCADE offrent plus de flexibilité, une flexibilité très utile pour la programmation.

**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. Celle-ci produit le flot `x` des entiers naturels qui "bégaie", c'est à dire, où chaque entier est répété deux fois. On peut ensuite utiliser 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 | ... |

On peut cependant critiquer le code ci-dessus. 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 `o`, qu'on peut comprendre comme une variante assez différente de `if c then e1 else e2`. Contrairement à la conditionnelle, la fusion de flots n'est pas point à point. Informellement, lorsque le prochain élément de $[\![c]\!]$ est vrai (resp. faux), la sortie de $[\![o]\!]$ 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**. 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 produits *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.

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

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

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

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

**Question 12.** *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` est 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 compilateur. Pour vous en convaincre, essayez de compiler le nœud précédent, en remplaçant `y when false(c)` par `y when c`.

```
$ 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 un plus conséquent qui les utilise de façon plus intéressante 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 13.** *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* $x_0, x_1, x_2, x_3, x_4$.

### *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 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

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.

n'est vraie, on reste dans l'état courant. Il existe des transitions de diverses sortes, que nous détaillerons ultérieurement.

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

[5] 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.

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

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