Informatique et sciences numériques

M. Gérard Berry, membre de l'Institut
(Académie des sciences) et de l'Académie des technologies,
professeur associé

Penser, modéliser et maîtriser le calcul informatique

Le cours « Penser, modéliser et maîtriser le calcul informatique » a été donné dans le cadre de la chaire annuelle d'informatique et sciences numériques. Il a été le tout premier cours de cette chaire, créée à la suite de mon cours précédent « Pourquoi et comment le monde devient numérique », lui-même donné en 2007-2008 dans le cadre de la chaire annuelle d'innovation technologique-Liliane Bettencourt. Le cours a été consacré à l'étude des fondements du calcul automatique, celui effectué par les ordinateurs petits ou grands présents dans tous les dispositifs numérisés, des méga-ordinateurs ou réseaux aux micro-puces embarquées dans les objets de toutes sortes. Comme je l'avais expliqué en 2007-2008, l'informatique est un art difficile si l'on veut faire la différence entre les applications qui marchent vraiment et celles qui marchent à peu près. Ces dernières se caractérisent par la présence de bugs résiduels induisant des comportements aberrants, avec des conséquences plus ou moins graves selon le type de système. Faire des applications qui marchent vraiment demande de *raisonner sur les calculs* qu'elles effectuent, à l'aide de modèles mathématiques, comme on raisonne sur des objets physiques. Cependant, à cause de la taille et de la complexité des circuits et programmes, on s'aperçoit rapidement que le raisonnement sur papier ne suffit pas. Il faut introduire la notion clef de *calcul sur le calcul*, pour faire effectuer ou au moins vérifier les raisonnements par les machines elles-mêmes, en visant la *vérification formelle* automatique ou assistée de programmes et de circuits. Mais, pour pouvoir travailler ainsi, il faut d'abord formaliser la notion de calcul elle-même, ou plutôt *les* notions de calcul tant la discipline a progressé récemment.

L'étude systématique du calcul est fort ancienne, puisqu'elle est déjà présente dans les traités des Indiens sur les nombres dès les premiers siècles de notre ère, et systématisée par Al Khuwarizmi au IX^e^ siècle. Mais il s'agissait ici du calcul humain. L'étude du calcul automatique a vraiment commencé avec Babbage et son assistance

Ada Lovelace, fille de Lord Byron, qui ont travaillé sur un prototype de machine mécanique, la machine à différences. Dès cette époque, et même si elle n'a jamais pu faire tourner ses programmes, Ada avait compris la difficulté intrinsèque de la programmation des machines automatiques. En logique, vers 1850, George Boole avait fait un pas majeur en montrant que la logique propositionnelle peut être résolue par calcul algébrique simple, reliant pour la première fois logique et algèbre[1]. L'étude vraiment formelle du raisonnement dans des logiques plus générales a commencé dans la seconde moitié du XIX^e^ siècle avec les travaux de Frege et des logiciens qui l'ont suivi.

Mais c'est dans les années 1930 que le sujet a vraiment explosé, avec les travaux d'Alan Turing et Alonzo Church sur la notion générale de calculabilité. Ils ont apporté deux découvertes cruciales : la thèse que la notion de calculabilité est indépendante du mécanisme de calcul (machine, système de réécriture, etc.), dite *thèse de Church-Turing*, et la découverte par Turing de l'existence de problèmes *indécidables* c'est-à-dire non solubles par calcul automatique. Deux formalismes majeurs du calcul ont été introduits lors de ces travaux : la *machine de Turing*, et le *λ-calcul* de Church. Ils sont d'égale puissance mais de style très différent. La machine de Turing est une machine abstraite sans utilité pratique, mais toujours utilisée comme outil de base pour les questions de complexité algorithmique et de sécurité informatique. Le *λ-calcul* est un calcul algébrique qu'on peut voir comme le plus ancien vrai langage de programmation. Il constitue toujours la base des *langages fonctionnels* comme ML, O'Caml, Scheme, Haskell, F#, Javascript, etc., qui prennent de plus en plus d'importance grâce à leurs qualités de rigueur et de simplicité. Il est extrêmement remarquable que machines de Turing et λ-calcul sont nés quasiment définitifs : chacun a été abondamment étudié et amélioré, avec par exemple l'introduction de systèmes de types sophistiqués en λ-calcul permettant la détection des bugs à la compilation, mais jamais les principes et notations initiales n'ont été remis en question. Cependant, ces objets sont de nature purement syntaxique. Dans les années 1970, Scott leur a donné une théorie sémantique fondamentale, basée sur la notion de croissance de l'information directement visible au cours d'un calcul. Beaucoup d'auteurs, dont moi-même, ont travaillé sur la relation syntaxe-sémantique, étudiée dans ce cours.

La notion *d'indécidabilité* apparaît immédiatement dans chacun de ces formalismes : l'arrêt du calcul d'une machine ou d'un λ-terme est indécidable, de même que l'égalité comportementale de deux machines de Turing ou λ-termes. Ceci vaut pour bien d'autres propriétés : machines de Turing et λ-calcul sont des formalismes de généralité maximale, où toutes les propriétés intéressantes sont indécidables. Dès les années 1950, et d'abord à cause de problèmes de linguistique, d'autres chercheurs se sont consacrés à l'étude de la classe des *systèmes d'états finis*, restreinte mais fondamentale de par ses applications. Comme leur nom l'indique,

1. Une autre invention d'Al Khuwarizmi, introduite dans son traité *Al Jabr* sur la solution des équations simples.

ces systèmes ne peuvent prendre qu'un nombre fini d'états de calcul, éventuellement très grand, au contraire des machines de Turing qui peuvent en avoir une infinité. On en trouve partout : dans l'analyse lexicale des textes, dans les protocoles de communication réseau, dans le contrôle et les organes de calcul des circuits électroniques, dans les interfaces homme-machine, dans le contrôle des systèmes de transports, etc. Au contraire des modèles généraux, toutes leurs propriétés sont décidables, et souvent efficacement. C'est le domaine où l'éradication des bugs est vraiment faisable, y compris dans les applications industrielles. De plus, leur théorie est remarquablement élégante et met à jour des concepts fondamentaux de portée plus générale, en particulier dans la relation entre langage et machine. Nous les étudierons donc avec soin.

Un point à noter est que tous les modèles précités sont de nature strictement *séquentielle* : le calcul se fait « avec un seul crayon », imitant le calcul humain traditionnel. Or, l'informatique moderne a depuis longtemps débordé du paradigme séquentiel en introduisant le *calcul parallèle* sous de nombreuses formes. D'abord, on a exécuté plusieurs programmes sur un seul ordinateur, à l'aide de systèmes d'exploitation ordonnançant l'exécution de ces programmes. Comme l'a bien exprimé Dijkstra dès les années 1950, pour raisonner sur cette exécution multiple sans dépendre des spécificités de l'exécution, il est naturel de voir les programmes comme s'exécutant logiquement en parallèle, même si ce n'est pas vrai physiquement. Puis les premiers réseaux ont permis de connecter des ordinateurs ; les protocoles de transmission utilisés à chaque bout doivent être vus comme des composants parallèles et distribués à partir d'une même application. Ensuite ont été développés les machines vectorielles ou multiprocesseurs, les grands réseaux d'ordinateurs, puis les circuits multicœurs. Il a fallu comprendre comment écrire des programmes pour ces machines et réseaux, avec deux options possibles : paralléliser automatiquement des programmes séquentiels classiques, pour les exécuter plus efficacement (ce qui demande encore une fois du calcul sur le calcul), ou développer de nouveaux langages parallèles adaptés à une programmation conceptuellement parallèle dès l'expression des algorithmes. La première méthode a eu et a toujours beaucoup de succès, alors que la seconde en a beaucoup moins, bien qu'elle paraisse intellectuellement plus satisfaisante : nous avons beaucoup de mal a « penser parallèle ». Ce n'est pas forcément surprenant, car les neuropsychologues nous apprennent que les mécanismes de l'attention et de la conscience sont profondément séquentiels[2].

Le cours a montré qu'il existe en fait trois parallélismes très différents les uns des autres. Dans le *parallélisme asynchrone*, les acteurs parallèles évoluent sans synchronisation directe et communiquent sans maîtrise du temps, soit en partageant des mémoires, soit en échangeant des messages. C'est le parallélisme des machines multiprocesseurs, du réseau Internet, ou des circuits multicœurs. On le trouve partout, mais, paradoxalement, c'est le plus difficile à maîtriser, tant au niveau de l'algorithmique qu'à celui de la programmation. Une des raisons est le caractère

2. Voir les cours de Stanislas Dehaene au Collège de France.

fondamentalement *non-déterministe* des systèmes asynchrones : produire deux fois les mêmes entrées ne produit généralement pas les mêmes sorties, à cause de l'ordre variable des communications. Bien que l'asynchronisme soit étudié très intensément, il reste théoriquement mal compris ; les modèles abondent, mais aucun ne se détache vraiment. Le cours en présente les plus importants, sans entrer dans les détails.

Dans le *parallélisme synchrone*, on suppose au contraire que la communication entre entités parallèles se fait en temps nul, comme pour la transmission de la gravité en mécanique newtonienne. Les systèmes qu'on considère alors sont les *systèmes réactifs*, qui réagissent immédiatement aux entrées provenant de leur environnement en fournissant des sorties vers cet environnement. Les systèmes réactifs synchrones sont dominants dans le domaine du contrôle de processus (pilotage d'avions, freinage de voiture, contrôle moteur, etc.), dans les interfaces homme-machines (cockpit d'avion, etc.), dans les jeux, etc. Beaucoup de ces systèmes sont critiques sur le plan de la sécurité. Au contraire du parallélisme asynchrone, le parallélisme synchrone est fondamentalement déterministe : les mêmes entrées produisent les mêmes sorties. Il est parfaitement compris théoriquement, ses mathématiques sont très élégantes, et les langages de programmation associés sont utilisés industriellement dans des applications critiques de grande taille (Airbus A380 pour SCADE par exemple ; *cf.* le cours de 2008, « Systèmes embarqués »).

La dernière sorte de parallélisme est le *parallélisme vibratoire*, dans lequel l'information se propage en temps prévisible et borné. C'est par exemple celui qui caractérise la transmission des fronts électriques dans les circuits électroniques. On peut le décliner sous de nombreuses formes, matérielles et logicielles. Il a d'importantes relations avec le parallélisme synchrone : dans un système assez compact, on peut « faire semblant » d'être synchrone en implémentant la fonctionnalité d'une façon vibratoire suffisamment rapide. On allie ainsi la simplicité conceptuelle du synchrone avec l'efficacité des implémentations vibratoires. Par exemple, dans un circuit électronique synchrone, le comportement est conceptuellement défini par un système d'équations booléennes synchrones, dont la solution est calculée dans un temps de l'ordre de la nanoseconde par la propagation des signaux électriques. Une correspondance analogue existe pour les implémentations logicielles des langages synchrones.

Dans les applications réelles, rien n'est aussi tranché et les trois formes de parallélisme doivent coopérer. Ce n'est pas sans poser des problèmes fondamentaux. Par exemple, les circuits modernes se composent de blocs synchrones bien compris communicant entre eux de façon asynchrone (GALS = *Globally Asynchronous Locally Synchronous*). Le cours montre la difficulté de cette composition, qui doit être vue de façon probabiliste à cause de phénomènes physiques incontournables de métastabilité des registres (échantillonneurs sur horloges) qui assurent la frontière asynchrone/synchrone. Enfin, le dernier séminaire d'Yves Frégnac dans ce cours, axé sur la neuro-informatique, a montré un type de coopération fort différent de modes de calcul dans le cortex visuel.

1. Conduite du cours

Le cours s'est composé de la leçon inaugurale et de huit séances. Les trois premières séances ont été consacrées à la théorie de la calculabilité et au λ-calcul, les trois suivantes aux différentes sortes de parallélisme, la dernière à la coopération entre les modes de calcul et aux réponses aux questions envoyées par courrier électronique. Chaque séance s'est composée de deux parties : le cours proprement dit et un séminaire donné par une ou plusieurs personnalités extérieures. Tous les cours et séminaires ont été mis en ligne en vidéo. Il faut noter qu'une trentaine d'étudiants de diverses universités se sont inscrits au cycle de cours qui a pu être validé pour leur formation.

2. Calculabilité et λ-calcul

Les trois premières leçons et les deux premiers séminaires ont été consacrés à la théorie de la calculabilité et au λ-calcul. Comme ces sujets sont abordés en détail dans le texte de la leçon inaugurale, nous n'en rappelons ici que l'architecture et les grands résultats.

2.1. Calculabilité

La théorie de la calculabilité s'intéresse à ce qu'il est possible ou impossible de faire avec une machine informatique. Elle est née des travaux de logiciens (Church, Gödel, Turing) dans les années 1930, soit bien avant la fabrication du premier ordinateur. Depuis les années 1970, elle se prolonge par la théorie de la complexité, qui analyse le coût d'un calcul faisable, en temps, mémoire ou énergie.

Le tout premier vrai modèle de calcul a été la *machine de Turing*, déjà décrite dans [Berry, 2008]. Cette machine fort simple est très robuste. Elle ne change pas de puissance en fonction de la taille de l'alphabet ni du nombre d'états (sous certaines contraintes minimales), quand on ajoute des bandes, qu'on la rend non-déterministe, *etc.* Bien qu'elle n'ait aucune application pratique, elle reste un outil fondamental en complexité des algorithmes et en sécurité informatique. De nombreux autres modèles équivalents ont été proposés. La *machine de Von Neumann* se compose d'une unité de calcul et d'une mémoire adressable, et est plus proche des ordinateurs réels. Les *systèmes d'équations récursives* définissent des fonctions par elles-mêmes, autorisant par exemple la définition équationnelle de la factorielle

$$\text{fact(n)} = \text{si n} = 0 \text{ alors } 1 \text{ sinon n*fact(n} - 1).$$

Le *λ-calcul* de Church donne une syntaxe très élégante et très puissante à la définition et à l'application (*cf.* 2.2). Encore dans un autre style, la classe des *fonctions récursives partielles* est la plus petite classe de fonctions entières partielles contenant les primitives arithmétiques de base (0, successeur, test à 0) et fermée par composition et prise du minimum $\mu x \cdot P(x,y)$, qui rend pour un y donné le plus petit x tel que P(x,y) est vrai, s'il existe. Des simulations réciproques montrent que tous ces formalismes ont la même puissance d'expression, et la *thèse de Church-Turing* soutient que cela restera vrai pour tous les formalismes à venir.

Le résultat le plus connu de la théorie de la calculabilité est *l'indécidabilité de l'arrêt d'une machine de Turing*. Il dit qu'on ne peut pas construire une machine de Turing décidant en temps fini si une machine de Turing donnée M (décrite par sa table de transition) s'arrête ou boucle sur une donnée D ; voir [Berry, 2008] pour la preuve. Ce résultat se généralise dans le *théorème de Rice*, qui exprime que toute propriété non-triviale de la classe des fonctions récursives partielle est indécidable. Ces résultats majeurs établissent les limites théoriques de la calculabilité, mais sont moins toxiques qu'il n'y paraît en pratique car les fonctions usuelles deviennent très difficiles à calculer bien avant d'être impossibles à calculer : il suffit qu'elles soient au moins exponentielles, d'où l'importance pratique de la théorie de la complexité.

Le cours a présenté d'autres concepts moins connus mais tout aussi essentiel, comme la hiérarchie stricte des fonctions primitives récursives, qui sont totales, et le théorème inexistence de formalismes définissant toutes les fonctions récursives totales et rien qu'elles.

2.2. Le λ-calcul

Le λ-calcul a été introduit par Church en 1936. C'est un calcul des fonctions apparemment tout simple, qui prend au sérieux la notation fonctionnelle classique $[x \to f(x)]$ en lui donnant un statut de première classe. Les variables sont notées x, y, z,… Le terme $\lambda x \cdot M$ dénote une fonction de x de corps M, et le terme MN décrit l'application d'un terme M à un terme N ; c'est tout pour la syntaxe. La loi de calcul est le remplacement du paramètre formel par l'argument réel dans l'application d'une fonction explicite à un argument : $(\lambda x \cdot M)N \to M[N/x]$. Une description plus précise est donnée dans [Berry, 2009] et la théorie syntaxique complète est donnée dans [Barendregt, 1987].

Le λ-calcul est un calcul parfaitement né, souvent complété mais jamais modifié en profondeur. Le cours a présenté ses grands théorèmes syntaxiques, décrits plus avant en [Barendregt, 1987] : les confluences locales et globales, qui expriment le déterminisme du calcul en tant qu'indépendance du résultat par rapport à l'ordre des calculs ; le théorème des développement finis généralisés de Lévy, qui met en lumière la causalité des réductions dans le λ-calcul et sert de marteau-pilon pour prouver bien d'autres résultats ; les théorèmes de stabilité et de séquentialité de l'auteur, qui montrent que le λ-calcul est fondamentalement incapable d'exprimer des calculs parallèles et reste de nature profondément séquentielle. Tous ces théorèmes sont difficiles et ont des preuves profondes.

Le λ-calcul pur est assez sauvage, car il permet des opérations contre-intuitives, comme l'application diagonale xx d'un objet x à lui-même. On peut le domestiquer en interdisant cette construction au moyen de systèmes de types, mais au prix d'une diminution de l'expressivité. Le cours a présenté les types ensemblistes standard de Church et les types polymorphes de ML, fondamentaux en programmation. Les transparents présentent aussi le système F de Girard, système de type particulièrement puissant pour définir une grande classe de fonctions

récursives totales. Ce système de types a été repris et étendu par Huet et Coquand dans la théorie des constructions, base du système CoQ de plus en plus utilisé en vérification de programmes et en mathématiques formelles. C'est en CoQ que Gonthier a donné la première vraie preuve (automatisée) du fameux théorème des quatre couleurs [Gonthier, 2008].

Le cours a également montré pourquoi le λ-calcul est de venue la base de nombreux et importants langages de programmation pratique, comme l'avait deviné Landin dans un papier historique [Landin, 1966] : ML/Caml, Haskell, Scheme , JavaScript, F#, etc.

2.3. Sémantique de Scott, stabilité et séquentialité

Le λ-calcul est un objet purement syntaxique qu'on utilise dans l'intention de définir des fonctions, donc des objets sémantiques. Ici se pose une difficulté : que peut vouloir dire une opération diagonale xx ? Elle demande l'identité des éléments d'un domaine D et de son espace fonctionnel $(D \to D)$, ce qui est impossible en théorie des ensembles classiques car la cardinalité de $(D \to D)$ est strictement plus grande que celle de D. Le problème a été résolu par Dana Scott, qui a introduit la notion d'ordre partiel complet (cpo) $<D, \subset, \perp>$ où $\subset$ est l'ordre d'information et $\perp$ l'information indéfinie, voir [Berry, 2009 ; Amadio et Curien, 1998] pour les détails. La sémantique de Scott a donné beaucoup de force au λ-calcul en permettant de le voir d'une façon différente, bien plus reliée aux intuitions programmatiques. Elle est utile dans de nombreuses applications, comme la vérification de programmes par interprétation abstraite présentée par Cousot dans un séminaire du cours 2008.

J'ai longtemps travaillé sur un problème essentiel du domaine : la recherche de la relation exacte entre la syntaxe et la sémantique du λ-calcul. Autrement dit, quelles sont les fonctions que l'on peut définir dans un langage fonctionnel ? En travaillant sur le langage PCF (*Programming Computable Functions*) qui est un noyau de langage fonctionnel fondé sur le λ-calcul typé, Plotkin avait montré que le modèle de Scott ne reflète pas la réalité syntaxique : si l'égalité de deux termes dans le modèle implique leur interchangeabilité dans les calculs syntaxiques, l'inverse est faux à cause de l'existence de fonctions de Scott non définissables en PCF, dont le prototype est la fonction « ou parallèle » pou(x,y) qui rend vrai dès qu'un de ses arguments est vrai, même si l'autre est indéfini. Cette fonction demanderait une évaluation parallèle des termes arguments, qui est impossible à cause des théorèmes de stabilité et de séquentialité mentionnés en 2.2. Milner a ensuite montré qu'il existait un modèle complètement adéquat unique, mais sans en donner la structure. Trouver cette structure était un des grands problèmes du domaine dans les années 1975-1990.

J'ai d'abord affiné la sémantique de Scott en imposant aux fonctions une condition de *stabilité*, qui exprime le déterminisme de la causalité de leurs calculs. Mais le modèle obtenu contient encore des fonctions non-définissables, dont le prototype est la « fonction de Gustave » (d'un de mes vieux surnoms), cf. [Berry, 2009].

Éliminer cette fonction a exigé la création et le développement d'une nouvelle théorie des algorithmes séquentiels, réalisés avec Pierre-Louis Curien. En reprenant des travaux de Kahn et Plotkin sur les domaines de Scott « concrets », nous avons pu définir un nouveau modèle séquentiel dont les objets sont des algorithmes et non plus de simples fonctions, ce qui a demandé une nouvelle vision catégorique des modèles du λ-calcul. Cette théorie n'a pas directement résolu le problème, et a été ensuite poursuivie dans un nouveau courant de sémantique de jeux (*game semantics*) par Abramsky *et al.*, toujours actif. Mais Curien a montré qu'elle était bien complètement adéquate pour le langage PCF augmenté d'un système d'exceptions proche de celui de Caml, ce qui est hautement satisfaisant en pratique. Le modèle des algorithmes séquentiels a aussi été implémenté dans le langage CDS, maintenant défunt, mais qui pourrait trouver de nouvelles applications.

L'ensemble de ces belles théories est présenté dans [Amadio et Curien, 1998], l'ouvrage de référence du sujet.

2.4. Séminaires sur le λ-calcul

Gérard Huet, directeur de recherches à l'INRIA, a présenté les relations entre λ-calcul et logique. Il a en particulier discuté l'utilisation du λ-calcul comme syntaxe pour les preuves en logique, ainsi que le rapport entre β-réduction et élimination des coupures en logique. Rassemblé sous le nom de *correspondance de Curry-Howard*, cet ensemble de résultats a profondément transformé la vision de la preuve de théorèmes sur ordinateur. Jean-Jacques Lévy, également directeur de recherches à l'INRIA et directeur du laboratoire commun INRIA-Microsoft, a approfondi la théorie syntaxique du λ-calcul en montrant les différences entre appel par valeur et appel par nom et en illustrant le passage du calcul théorique aux langages de programmations fonctionnels modernes.

3. Systèmes d'états finis

Comme leur nom l'indique, les systèmes d'états finis sont caractérisés par le fait qu'ils ne peuvent prendre qu'un nombre fini d'états. Cela rend toutes leurs propriétés décidables, ce qui est très intéressant en pratique. Mais ces systèmes peuvent être très grands, et l'algorithmique pour vérifier réellement leurs propriétés est particulièrement délicate. Comme elle est très importante en pratique, y compris en contexte industriel (circuits, avionique, protocoles de communication, etc.), elle a fait beaucoup de progrès ces dernières années. Sur ce sujet, nous renvoyons le lecteur à la séance 5 (vérification) du cours 2007-2008. Nous nous consacrons ici aux modèles de calcul associés, de façon assez détaillées car ce sujet n'a pas été abordé dans le texte de la leçon inaugurale [Berry, 2009]. La vidéo du cours montre évidemment les choses de façon plus animée.

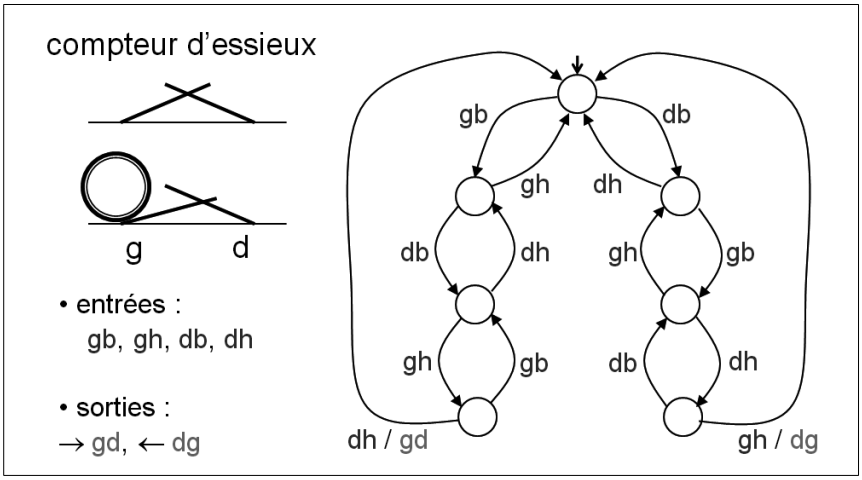

Figure 1 : **le compteur d'essieux**

Deux formalismes sont ici fondamentaux : les diagrammes d'états, et en particulier les *automates finis*, et les descriptions linguistiques, en particulier les *expressions régulières*. Ces deux formalismes ont la même puissance mais sont reliés par une dualité non-triviale. Nous nous concentrons ici sur leurs traductions réciproques.

La figure 1 montre un exemple intéressant d'automate fini. Il s'agit d'un compteur d'essieux de trains, composé de deux pédales g (gauche) et d (droite), chacune envoyant un signal quand elle atteint la position basse (gb, db) ou haute (gh, dh). Ces signaux sont les entrées de l'automate, dont l'état initial est indiqué par une petite flèche. Quand une roue du train passe de gauche à droite, la séquence est gb–db–gh–dh, et l'automate envoie la sortie gd pour « gauche-droite ». La situation est symétrique pour un passage de droite à gauche, avec la sortie dg. Mais la roue peut aussi osciller sur les pédales, provoquant alors des retours arrière dans l'automate. Ici, le comportement est par essence infini, et l'automate n'a pas d'état final.

Un autre exemple est fourni dans la figure 2. On y voit deux automates, chacun possédant un état initial et un état final indiqué par le double cercle[3]. La légende fait apparaître l'expression rationnelle (ab + b)*ba. Dans les deux cas, les entrées sont les lettres a et b, et la sortie est simplement l'acceptation d'un mot sur {a,b}. L'expression rationnelle exprime que les mots acceptés sont composés d'un nombre quelconque d'occurrences de « ab » ou « b » suivi d'une occurrence de « ba ». Dans l'automate, un mot est reconnu par un automate si et seulement il conduit de l'état

3. En général, un automate a un état initial et un nombre quelconques d'états terminaux.

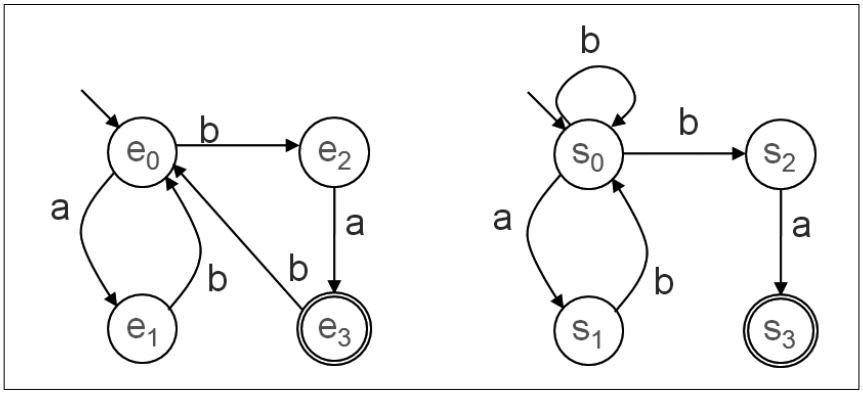

Figure 2 : **automates pour (ab + b)*ba**

initial à un état final en suivant des flèches étiquetées par ses lettres successives. Par exemple, « abba », « bba », et « abbbbabba » sont reconnus, alors que « bab » ne l'est pas. L'automate de gauche est *déterministe*, i.e. est tel qu'il n'y a jamais deux flèches partant d'un même état et portant la même lettre. En revanche, l'automate de droite est *non-déterministe* : de l'état initial partent deux flèches étiquetées b, et il faut choisir le bon chemin pour reconnaître un mot. Ces deux sous-classes d'automates finis sont toutes deux essentielles, pour des raisons différentes : le déterminisme rend l'automate apparemment plus simple, mais peut le faire exploser en taille ; le non-déterminisme supprime cet inconvénient, mais est plus subtil à étudier et à implémenter.

3.1. Syntaxe des expressions régulières

Étant donné un alphabet $X = \{a,b,c,\ldots\}$, le *monoïde libre* X^* est formé des mots écrits sur X ; la *concaténation* de deux mots u et v les met bout à bout et engendre uv ; le *mot vide* (sans lettre) est noté ε. Un *langage* $L \subset X^*$ est un ensemble de mots sur X. On construit les *expressions régulières* e, e', etc. sur X et on détermine leurs *langages engendrés* L(e), L(e'), etc., de la façon suivante :

– 0 est l'expression vide, avec $L(0) = \varnothing$;
– 1 est l'expression mot vide, avec $L(1) = \{\varepsilon\}$;
– a est l'expression lettre, avec $L(a) = \{a\}$;
– $e + e'$ est la *somme* de e et e', avec $L(e + e') = L(e) \cup L(e')$;
– $e \cdot e'$, ou simplement ee', est le *produit* de e et e', avec $L(e \cdot e'') = \{uu', u \in L(e), u' \in L(e')\}$;
– e^* est l'itérée (ou étoile) de e, avec $L(e^*) = \{u_0\ u_1,\ldots\ u_n \mid n \geqslant 0,\ u_n \in L(e)\}$. Notons que le cas $n = 0$ entraîne $\varepsilon \in L(e^*)$.

Un langage est dit *rationnel* (ou *régulier*, *regular* en anglais) s'il peut être engendré par une expression rationnelle.

Écrivons simplement e = e' si et seulement si e et e' engendrent le même langage, i.e., L(e) = L(e'). Les égalités suivantes sont immédiates : $e + 0 = 0 + e = e$, $e \cdot 1 = 1 \cdot e = e$, $e \cdot (e' + e'') = e \cdot e' + e \cdot e''$, $(e + e') \cdot e'' = e \cdot e'' + e' \cdot e''$, $e^* = 1 + e \cdot e^*$. On définit le prédicat « engendre le mot vide » $\varepsilon(e) \in \{0,1\}$ par $\varepsilon(0) = 0$, $\varepsilon(1) = 1$, $\varepsilon(a) = 0$, $\varepsilon(e + e') = \varepsilon(e) + \varepsilon(e')$, $\varepsilon(e \cdot e') = \varepsilon(e) \cdot \varepsilon(e')$, $\varepsilon(e^*) = 1$. Ici, 0 et 1 sont des expressions rationnelles jouant le rôle de booléens à l'intérieur du formalisme, et on applique les simplifications de base sur 0 et 1 pour toujours obtenir 0 ou 1 en résultat.

D'autres opérateurs comme la négation $\neg$ ou l'intersection $\cap$ peuvent être ajoutés aux expressions rationnelles. Nous verrons qu'ils ne modifient pas leur puissance d'expression, mais sont difficiles à implémenter efficacement.

3.2. D'un automate à une expression rationnelle

Pour montrer qu'automates et expression rationnelle ont la même puissance d'expression, on construit une traduction systématique d'un automate en expression rationnelle. Nous n'étudierons pas ici cette traduction, indiquant simplement qu'elle fait appel à un calcul matriciel de fermeture transitive, en partant d'une matrice carrée dont les lignes et colonnes sont associées aux états, avec comme valeur les expressions rationnelles faisant passer d'un état à un autre.

Le point important est que l'expression rationnelle ainsi associée à un automate peut être de *taille exponentielle* dans la taille de l'automate, et que ceci peut être vrai pour toute expression rationnelle engendrant ce langage. Les expressions rationnelles peuvent donc être exponentiellement plus succinctes que les automates. L'inverse peut être également vrai, ce qui rend la dualité hautement non-triviale.

3.3. De l'expression rationnelle à un automate déterministe

Il existe plusieurs algorithmes pour traduire des expressions rationnelles en automate. Nous étudions ici notre préféré, dit *algorithme de Brzozowski* [Berry et Sethi, 1986]. Il est fondé sur la notion de *dérivée* d'une expression rationnelle par une lettre. Considérons une expression e qui peut écrire a comme première lettre. L'idée de la dérivation de par une lette a est de construire l'expression $a^{-1}(e)$ qui écrit le mot m ssi e écrit le mot am. Par exemple, $a^{-1}((ab + b)^*ba) = b(ab + b)^*ba$ puisque a ne peut être écrit en première lettre qu'à partir de ab, et $b^{-1}((ab + b)^*ba) = (ab + b)^*ba + a$, car il y a deux façons de commencer par b : soit utiliser le b de droite dans ab + b en développant une fois l'étoile, soit développer 0 fois l'étoile et écrire le b de ba. Les formules générales pour la dérivation sont les suivantes :

- $a^{-1}(0) = 0$;
- $a^{-1}(1) = 0$;
- $a^{-1}(a) = 1$;
- $a^{-1}(b) = 0$ si $b \neq a$;
- $a^{-1}(e + e') = a^{-1}(e) + a^{-1}(e')$;
- $a^{-1}(e \cdot e') = a^{-1}(e) \cdot e' + \varepsilon(e) \cdot a^{-1}(e')$;
- $a^{-1}(e^*) = a^{-1}(e) \cdot e^*$.

On suppose appliquées les simplifications de base décrites ci-dessus pour enlever les 0 et 1 inutiles. On généralise la dérivation par une lettre à la dérivation par un mot quelconque en posant $\varepsilon^{-1}(e) = e$ et $(au)^{-1}(e) = u^{-1}(a^{-1}(e))$, c'est-à-dire en dérivant successivement de gauche à droite par toutes les lettres d'un mot u pour obtenir $u^{-1}(e)$. Calculons et nommons toutes les dérivées itérées distinctes de notre expression favorite :

- $e_0 = e = (ab + b)^*ba$;
- $a^{-1}(e_0) = b(ab + b)^*ba = e_1$;
- $b^{-1}(e_0) = (ab + b)^*ba + a = e_2$;
- $a^{-1}(e_1) = 0$;
- $b^{-1}(e_1) = (ab + b)^*\ ba = e_0$;
- $a^{-1}(e_2) = b\ (ab + b)^*\ ba + 1 = e_3$;
- $b^{-1}(e_2) = (ab + b)^*\ ba + a = e_2$;
- $a^{-1}(e_3) = 0$;
- $b^{-1}(e_3) = (ab + b)^*\ ba = e_0$;

En oubliant le texte des expressions intermédiaires e_i et en ignorant les dérivées nulles, le système se réduit à $a^{-1}(e_0) = e_1$, $b^{-1}(e_0) = e_2$, $a^{-1}(e_1) = e_2$, $b^{-1}(e_1) = e_0$, $a^{-1}(e_2) = e_3$, $b^{-1}(e_2) = e_2$, $b^{-1}(e_3) = e_0$. On lit alors une équation $a^{-1}(e_0) = e_1$ comme une transition d'états $e_0 \text{–a}\rightarrow e_1$ dans un automate dont les états sont les e_i, ce qui donne directement l'automate de gauche de la figure 2. Le *théorème de Brzozowski*, que nous ne démontrerons pas ici, exprime que le nombre de dérivées itérées distinctes d'une expression rationnelle est fini, ce qui prouve que la méthode permet de traduire toute expression rationnelle en automate fini. Cependant, l'automate obtenu peut être de taille exponentielle dans la taille de l'expression rationnelle. Nous ferons mieux par la suite en utilisant des automates non-déterministes.

3.4. Négation et intersection

Notons immédiatement que l'on peut ajouter la négation et l'intersection comme opérateurs d'expressions rationnelles, avec comme lois ensemblistes pour les langages engendrés la complémentation $L(\neg e) = \neg L(e)$ et l'intersection $L(e \cap e') = L(e) \cap L(e')$. Les lois de dérivation restent simplement linéaires, avec $a^{-1}(\neg e) = \neg a^{-1}(e)$ et $a^{-1}(e \cap e') = a^{-1}(e) \cap a^{-1}(e')$, et la finitude de l'espace des dérivées formelles reste vraie. On peut donc construire un automate fini déterministe à partir de toute expression régulière contenant $\neg$ et $\cap$ exactement comme précédemment, ce qui prouve que *la classe des langages réguliers est fermée par complémentation et intersection* : étant donné deux expressions standard e et e', on peut construire l'automate pour $e \cap e'$, puis une expression régulière standard à partir de cet automate (cf. 2.2).

Cependant, négation et intersection sont très efficaces pour provoquer une explosion exponentielle des automates et doivent être maniées avec soin.

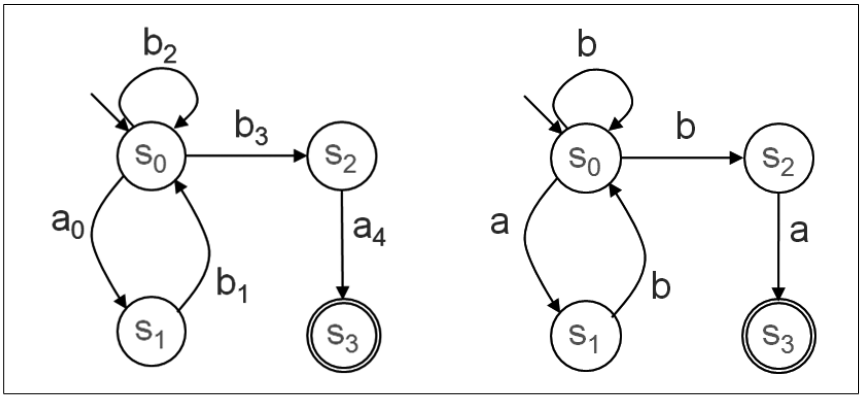

Figure 3 : **utilisation du langage linéarisé**

3.5. Expressions linéaires et automates non-déterministes

Présentons maintenant une construction bien plus efficace pour les expressions régulières standard, sans ¬ ni ∩, présentée en [Berry et Sethi, 1986]. L'idée centrale est que *le calcul des dérivées peut être rendu bien plus simple si l'expression est linéaire*, i.e., si elle est telle que chaque lettre n'apparaît qu'au plus une fois. Notre expression (ab + b)*ba n'est pas linéaire, mais on peut trivialement la linéariser en indexant les lettres pour les différencier, ce qui donne l'expression $s_0 = (a_0b_1 + b_2)^*b_3a_4$. Les dérivées par rapport aux lettres indexées deviennent beaucoup plus simples :

- $a_0^{-1}(s_0) = b_1(a_0b_1 + b_2)^*\ b_3a_4 = s_1$;
- $b_2^{-1}(s_0) = (a_0b_1 + b_2)^*\ b_3a_4 = s_0$;
- $b_3^{-1}(s_0) = a_4 = s_2$;
- $b_1^{-1}(s_1) = s_0$;
- $a_4^{-1}(s_0) = \varepsilon = s_3$.

L'automate déterministe pour les lettres indexées obtenu est celui de gauche dans la figure 3. En enlevant les indices, on obtient l'automate non-déterministe de droite pour les lettres non-indexées, déjà vu dans la figure 2, qui reconnaît le langage engendré par l'expression initiale (ab + b)*ba.

La linéarisation assure une propriété essentielle des dérivées *: les dérivées non nulles d'une expression linéaire de la forme* $(ua)^{-1}(e)$ *ne dépendent que de la lettre a et sont indépendantes du préfixe u.* Ceci se démontre par récurrence sur la taille d'une expression linéaire e. Par exemple, pour $(ua)^{-1}(e + e')$ non nulle, la lettre doit apparaître soit dans e soit dans e', mais pas dans les deux. On a donc soit $(ua)^{-1}(e + e') = (ua)^{-1}(e)$ si a apparaît dans e, soit $(ua)^{-1}(e + e') = (ua)^{-1}(e')$ si a apparaît dans e'. Notons que cette propriété n'est pas vraie si on utilise la négation ou l'intersection dans les expressions.

Ceci implique que, pour e linéaire contenant n lettres, l'automate des dérivées de e a au plus n + 1 états, en comptant l'état initial e. De plus, les flèches et états terminaux se déterminent simplement :

– il y a une flèche de l'état e vers l'état $a^{-1}(e)$ si et seulement si e peut écrire a comme première lettre ;

– il y a une flèche d'un état $(ua)^{-1}(e)$ vers un état $(uab)^{-1}(e)$ si b peut suivre a dans L(e), i.e., s'il existe un mot de la forme vabw dans L(e) ;

– un état est terminal ssi sa lettre peut être écrite comme dernière lettre d'un mot.

On n'a alors plus besoin de calculer explicitement les dérivées ; il suffit de calculer deux fonctions first(e) et last(e) donnant les ensembles de premières et dernières lettres écrites par e, ainsi qu'un prédicat follow(e, x, y) ∈ {0,1} testant si une lettre x peut suivre une lettre y dans un mot de L(e). Voici des formules simples pour calculer first et follow (d'autres encore plus efficaces peuvent être trouvées en [Berry et Sethi, 1986]) :

first (0) = first(1) = ∅	last (0) = last(1) = ∅
first (a) = {a}	last (a) = {a}
first(e + e') = first(e) ∪ first(e')	last(e + e') = last(e) ∪ last(e')
first(ee') = first(e) + ε(e) × first(e')	last(ee') = last(e) × ε(e') + last(e')
first(e*) = first(e)	last(e*) = last(e)

follow(0, x, y) = follow(1, x, y) = follow(a, x, y) = 0
follow(e + e', x, y) = follow(e, x, y) + follow(e', x, y)
follow(e · e', x, y) = follow(e, x, y)
+ follow(e', x, y)
+ (x ∈ last(e) × y ∈ first(e'))
follow(e*, x, y) = follow(e, x, y)
+ (x ∈ last(e) × y ∈ first(e))

Ces formules permettent un calcul très rapide des fonctions first, last et follow, ce qui permet d'écrire immédiatement l'automate de la figure 4 pour notre expression préférée. Cet automate a la propriété d'être *1-local* : toutes les flèches portant une même lettre convergent sur le même état. Ici, on s'aperçoit immédiatement que l'état initial est équivalent à l'état d'arrivée des flèches b_2, car il a les mêmes successeurs ; on peut donc l'enlever et déclarer initial l'état du milieu.

En enlevant enfin les indices, on obtient l'automate non-déterministe de la figure 5, qui reconnaît (ab + b)*ba. Il est moins compact que celui de la figure 2, mais obtenu à coût quasi nul. Son nombre de sommets borné par n + 1 si l'expression e contient n occurrences de lettres, et il a au plus n^2 flèches. On évite donc complètement toute explosion exponentielle.

3.6. Déterminisation

Pour passer d'un automate non-déterministe à un automate déterministe, on utilise un *algorithme de déterminisation* très simple. Un état de l'automate déterministe résultant est un ensemble d'états de l'automate non-déterministe,

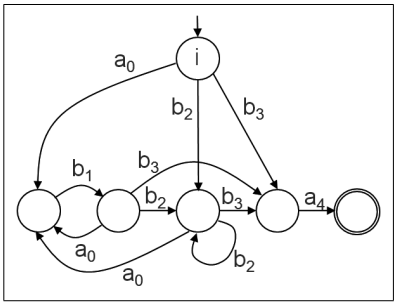

Figure 4 : **résultat de first, last et follow**

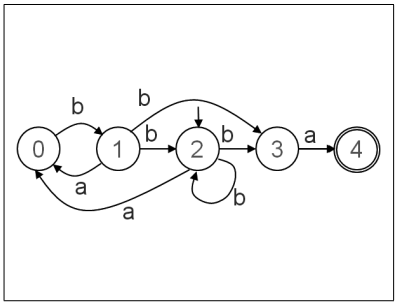

Figure 5 : **automate non-déterministe final pour (ab + b)*ba**

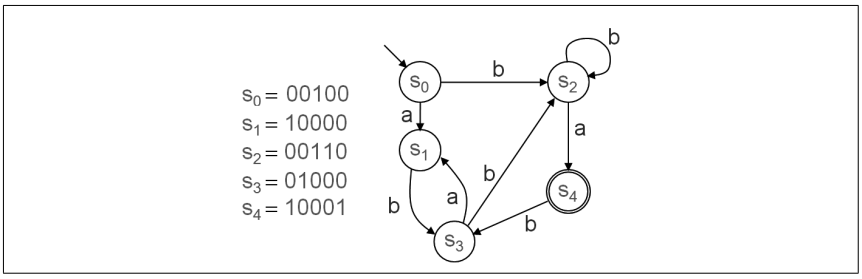

Figure 6 : **automate déterminisé**

partant du singleton de l'état initial. Il y a une flèche d'un ensemble vers un autre si et seulement si un état du premier ensemble a une flèche vers un état du second. Un état du déterministe codé par un ensemble est terminal si et seulement si il contient un état terminal du non-déterministe.

Partant de l'automate de la figure 5, on obtient ainsi l'automate déterministe de la figure 6. Chaque état s_i y est défini par sa fonction caractéristique sur les états de la figure 5, par exemple s2 = {2,3}. Les ensembles ne faisant pas partie de la liste sont inaccessibles.

La déterminisation est une opération dangereuse car elle peut faire exploser exponentiellement la taille d'un automate. Nous verrons en 5.2 comment l'éviter en compilant l'automate en circuit booléen.

3.7. Minimisation

L'automate de la figure 6 est plus gros que celui de gauche dans la figure 2. Mais il est facile de voir que ses états s_0 et s_3 sont équivalents, et de le minimiser. De manière générale, il existe toujours un automate déterministe minimal (en nombre d'états) unique reconnaissant un langage rationnel L ; ses états sont simplement les

langages suffixes de L, c'est-à-dire les langages L' définis par $\exists u.\ v \in L' \Leftrightarrow uv \in L$. Il existe un algorithme très rapide (en $n \cdot \log(n)$) pour calculer l'automate minimal, cf. [Sakarovitch, 2003].

L'algorithme de Berry-Sethi peut engendrer des automates bêtement trop gros. De nombreuses améliorations efficaces en ont été proposées (voir le travail de thèse tout récent de Benoît Razet sur les machine d'Eilenberg effectives : [Razet, 2009]). Enfin, signalons qu'il n'existe en général pas d'automate non-déterministe minimal pour un langage donné.

3.8. Suites automatiques et nombres transcendants

La leçon sur les automates finis s'est terminée par l'énoncé d'un magnifique et surprenant résultat montrant que les automates permettent de produire explicitement un grand nombre (dénombrable) de nombres réels transcendants, ce qui n'est pas commun. Considérons un automate A sur l'alphabet {0,1} déterministe et complet, c'est-à-dire tel que de tout état part une transition 0 et une transition 1. Pour chaque état, définissons un bit de sortie 0 ou 1 (pas besoin d'état final). Définissons une suite a_n, $n > 1$, de la façon suivante : on écrit n en binaire, par exemple $13 \rightarrow 1101$, et on passe la suite de 0 et de 1 obtenus dans l'automate, de gauche à droite ; alors a_n est le bit porté par l'état obtenu à la fin de cette suite de transitions. Le résultat central (obtenu par conjonction de théorèmes dus à Cobham et à d'Adamczewski et Bugeaud) exprime que le nombre réel $0{,}a_1 a\ a_3\ldots$ est transcendant.

La figure 7 montre deux exemples. Celui de gauche définit le nombre réel 0,10100100001… où chaque 1 est suivi d'un nombre de 0 double à chaque fois, donc grandissant exponentiellement. L'exemple de droite montre l'automate dit *du pliage de papier*, étudié en cours. Prendre un papier et le plier de gauche à droite, reprendre le papier plié de gauche à droite, et recommencer autant de fois qu'on veut. Déplier le papier. Lire de bas en haut les virages, en posant 1 = gauche, 0 = droite. La suite grandit de façon monotone avec le nombre de pliages, pour donner le réel 0,1101100111001 00... Elle est engendrée par l'automate de droite.

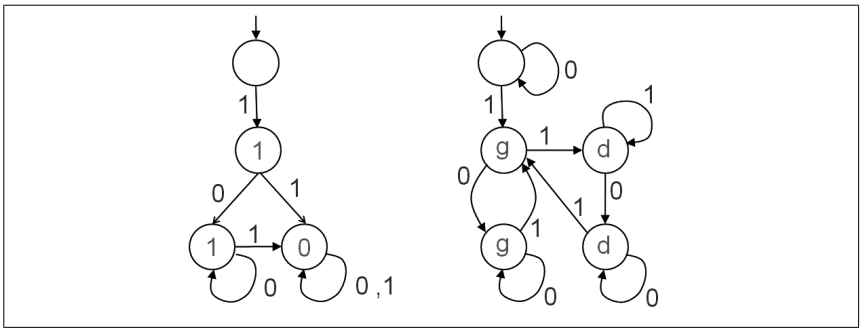

Figure 7 : **automates de Liouville et du pliage de papier**

Les deux nombre réels engendrés sont donc transcendants car trivialement non rationnels. Pour les détails, voir les vidéos et transparents, et pour tout savoir sur ces merveilleuses *suites automatiques* : [Allouche et Shallit, 2003].

4. Parallélisme asynchrone

Le parallélisme asynchrone est caractérisé par le fait que les différents acteurs d'un calcul n'ont pas de base de temps commune. Il se divise en plusieurs sous-catégories, en fonction de trois critères : comment les acteurs communiquent, comment ils coopèrent, et comment sont réglées leurs compétitions pour l'accès à des ressources communes. Il conduit naturellement au non-déterminisme des calculs. De nouveaux types de bugs y apparaissent immédiatement, les plus importants étant les suivants :

– l'interblocage ou *deadlock* : deux processus ont besoin de deux ressources communes et chacun en détient une sans vouloir la lâcher ;

– la famine ou *starvation* : plusieurs processus ont un besoin constamment renouvelé d'une ressource, mais l'un d'entre eux ne peut l'avoir qu'un nombre fini de fois au cours d'une exécution qu'il souhaiterait pourtant infinie.

4.1. La mémoire partagée

C'est l'extension la plus simple et apparemment la plus naturelle de la programmation séquentielle classique. Plusieurs processus séquentiels partagent une mémoire commune, qui règle les conflits d'écriture en autorisant seulement un processus à la fois à écrire dans une case mémoire donnée. Ce parallélisme se trouve en de nombreuses circonstances : gestion de processus par un système d'exploitation, circuits multicœurs, gestion d'affichages et d'interaction dans un navigateur Web, gestion de bases de données centralisées ou distribuées, etc. La difficulté est celle de la cohérence du non-déterminisme. Considérons l'exemple tout simple de la mise en parallèle P || Q de deux processus P et Q travaillant sur une variable X initialisée à 1 et définis ainsi :

P : X := X + 1
Q : local Y in Y := X ; X := 2*Y end

où Y est une variable locale à Q. Exécuter P avant Q revient à exécuter la séquence

X := X + 1 ; Y := X ; X := 2*Y

qui produit X = 4. Exécuter Q avant P revient à exécuter la séquence

Y := X ; X := 2*Y ; X := X + 1

qui produit X = 3. Ces deux exécutions sont compréhensibles et souhaitables, donc il est normal de dire que le résultat de l'exécution de P || Q peut être 2 ou 3.

Mais, si on laisse faire la nature, un interclassement des instructions de P et Q est aussi possible :

$$Y := X\ ;\ X := X + 1\ ;\ X := 2{*}Y$$

ce qui produit alors la valeur inattendue $X = 2$, l'addition étant ignorée. Cette valeur doit en général être rejetée car elle résulte d'une *interférence* entre P et Q ; c'est à cause de ce type d'interférence que deux voyageurs pouvaient encore récemment se trouver avec une réservation pour la même place de train, mésaventure toujours désagréable.

Pour éviter l'interférence, il faut demander à P et Q de s'exécuter au moins conceptuellement de façon *atomique*, c'est-à-dire pas du tout ou entièrement. Ceci peut se faire de plusieurs façons, la plus simple étant la pose de verrous ou *locks*. Un verrou ne peut être acquis que par un processus à la fois, un processus ne peut avancer qu'en ayant acquis le verrou, et il doit le rendre à la fin de son exécution. En supposant qu'on sait implémenter un tel verrou partagé L, ce qui peut se faire de plusieurs manières dont aucune n'est vraiment simple, on transforme P et Q ainsi :

P : lock(L) ; X := X + 1 ; unlock(L)
Q : lock(L) ; local Y in Y := X ; X := 2*Y end ; unlock(L)

Mais les verrous posent des problèmes très délicats : si un processus oublie de rendre un verrou, il bloque tous ceux qui utilisent ce verrou ; s'il faut verrouiller plusieurs ressources entre plusieurs processus, la gestion des verrous devient un cauchemar ; si les calculs d'un processus sont longs, en verrouiller l'intégralité introduit une inefficacité inacceptable. Les vraies solutions pratiques sont donc bien plus complexes que ce que nous avons présenté ici et il est fort difficile de prouver qu'elles sont correctes. Voir [Herlihy et Shavit, 2009] pour une analyse exhaustive du problème et des algorithmes pratiques.

4.2. Les réseaux de Kahn

À l'opposé absolu de la mémoire partagée générale, on trouve les réseaux dits de flots de données ou *data-flow*. Le plus bel exemple en est celui des *réseaux de Kahn* [McQuenn, 2009], qui en a donné la très élégante sémantique mathématique au début des années 1970 en utilisant la théorie de Scott (cf. 2.3). Comme montré dans la figure 8, un réseau de Kahn se compose de nœuds reliés par des files d'attente fifo (*first-in first-out*), chaque fifo ayant un nœud source et un nœud cible, plus des entrées et des sorties. Chaque nœud est un programme séquentiel déterministe classique augmenté par des instructions d'écriture dans ses fifos de sortie et de lecture dans ses fifos d'entrée. L'écriture est non bloquante, c'est-à-dire sans condition, mais la lecture est suspendue dans le cas où la fifo d'entrée est vide. Les nœuds sont exécutés de façon asynchrone, sans stratégie particulière. Cela implique qu'une fifo peut grandir de façon non bornée si son écrivain écrit plus souvent que son lecteur ne lit.

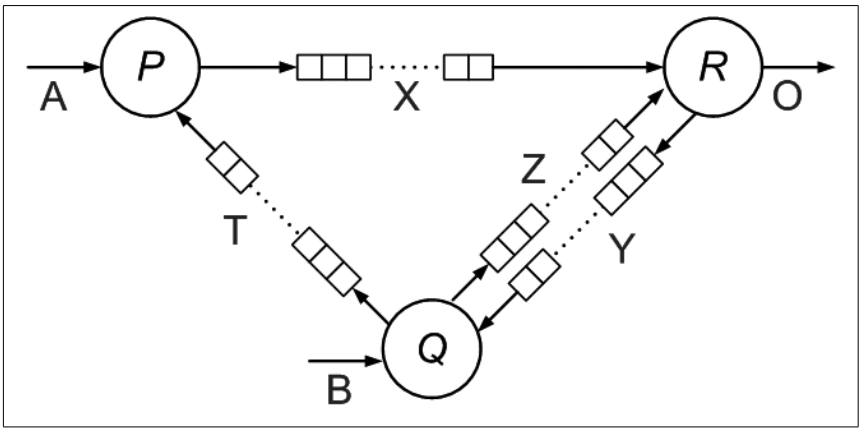

Figure 8 : **réseau de Kahn**

Les fifos servent à supprimer toute interférence en découplant totalement l'émission et la réception d'une valeur, donc en rendant les liens de causalité intemporels. L'idée de la sémantique de Kahn est de calculer les suites finies ou infinies de valeurs produites lors des exécutions du réseau. Dans le cas de la figure 8, il est facile de voir que ces suites doivent être solution du système d'équations suivant où les opérateurs des nœuds sont continus au sens de Scott, en considérant l'ordre préfixe des suites finies ou infinies :

$$X = P(A,T)$$
$$(O,Y) = R(X, Z)$$
$$T = Q(B,Y)$$

La théorie de Scott dit alors immédiatement qu'une solution minimale unique existe, et donc que le réseau est globalement déterministe, au sens où toutes les exécutions complètes équitables (où tout nœud qui le demande est exécuté infiniment souvent) fournissent le même résultat pour chaque fifo.

Les réseaux de Kahn sont d'utilisation constante dans le traitement du signal audio ou vidéo, dans la transmission Internet (pour le *streaming*), ainsi que dans la recherche d'information dans les grandes masses de données (*data mining*). Ils sont aussi la base des filtres d'Unix. On leur donne souvent des contraintes additionnelles comme des bornes sur la taille des fifos. Ces contraintes doivent être maniées avec le plus grand soin car elles cassent très facilement la sémantique mathématique. Le système Ptolemy d'Ed Lee et ses collaborateurs à Berkeley les étudie systématiquement et étudie d'autre versions du flot de données.

Une option tout à fait opposée a été proposée par Hoare dans le langage CSP, puis reprise par Milner dans la série de calculs de processus asynchrones commençant avec CCS [Milner, 1980]. Ici, l'interférence est supprimée par couplage fort de

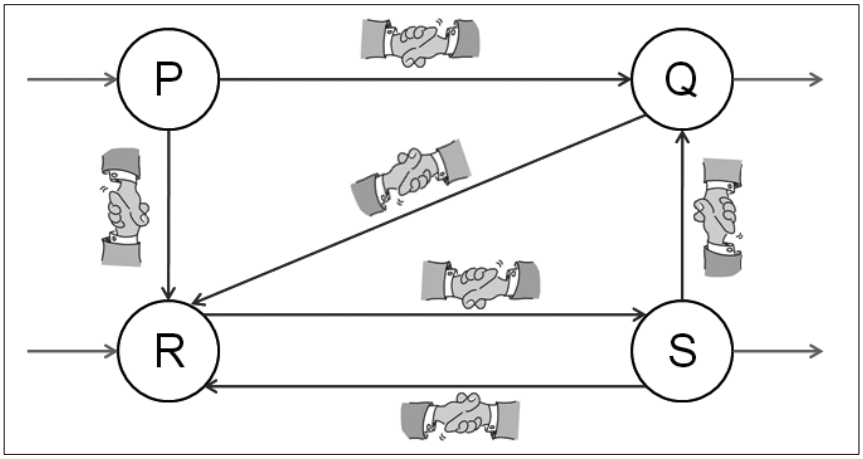

Figure 9 : **communication par rendez-vous**

deux processus qui veulent échanger une information. Ces processus se « serrent la main » dans un acte mutuel de rendez-vous, en échangeant éventuellement des valeurs (cf. figure 9). Chaque rendez-vous se produit sur un canal nommé et exclusif entre les deux processus. L'avantage est que *chacun sait ce que sait l'autre* à ce moment, contrairement au cas de la mémoire partagée. Dans le cas général, on utilise plusieurs processus et plusieurs canaux.

Des sémantiques mathématiques sophistiquées ont été élaborées pour ce modèle. La notion centrale, due à Milner, est la *bisimulation*. Deux processus sont bisimilaires s'ils peuvent faire les mêmes actions et se transformer par cette action en processus récursivement bisimilaires. Il faut noter que la simple sémantique de traces (mots de calcul) n'est pas pertinente pour les processus interactifs non-déterministes. Considérons par exemple deux distributeurs de boissons définis en CCS

– M1 = €. (Café.VerserCafé.M1 + Thé.VerserThé.M1) ;
– M2 = €.Café.VerserCafé.M2 + €.Thé.VerserThé.M2.

La machine M1 accepte un euro, puis un appui soit sur la touche Café soit sur la touche Thé, qui lui fait verser la boisson correspondante ; elle se régénère alors. La machine M2 accepte un euro, puis choisit elle-même de façon non-déterministe de n'accepter que la touche Café ou la touche Thé, sans que l'utilisateur n'y puisse rien. Vues en termes de traces, ces deux machines sont identiques, réalisant des suites finies ou infinies des mots Café.VerserCafé et Thé.VerserThé. Mais vues de l'utilisateur et de la bisimulation elles sont différentes : après €, M1 accepte Café et Thé, alors que M2 accepte soit Café soit Thé mais pas les deux ; les machines ne sont donc pas bisimilaires. La bisimulation a des propriétés algébriques fondamentales (cf. [Milner, 1980]).

Sur le plan pratique, le rendez-vous a servi de base aux langages de programmations CSP et OCCAM, ainsi qu'à ADA, qui couple rendez-vous et files fifo de manière

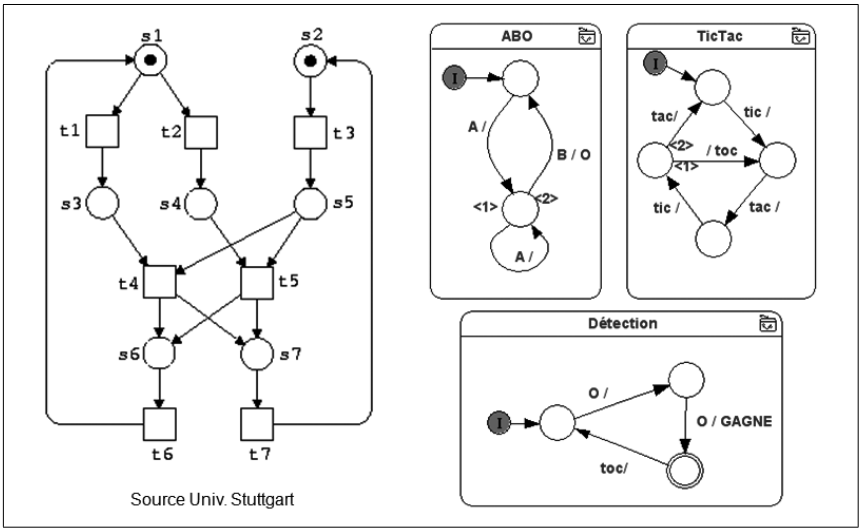

Figure 10 : **formalismes graphiques**

assez complexe. Il sert aussi de base au calcul de processus LOTOS utilisé dans le domaine des protocoles de communication. Mais le rendez-vous s'avère trop lourd pour la programmation parallèle générale, et son influence décline.

4.4. Modèles graphiques

Même si elle est techniquement moins commode qu'une représentation algébrique, une représentation graphique est souvent plus parlante pour l'utilisateur. La figure 10 montre deux formalismes classiques, les réseaux de Petri et les réseaux d'automates. Ils ne seront pas étudiés ici faute de place. Les termes CCS et Lotos sont aussi souvent représentés graphiquement (cf. [Milner, 1980]).

4.5. La machine chimique

La machine chimique ou CHAM a été introduite par moi-même et G. Boudol [Berry et Boudol, 1992]. Les processus y sont représentés par des molécules portant des valences d'interaction, nageant conceptuellement dans une solution, et pouvant librement se rencontrer à tout moment ; techniquement, cela s'exprime par de la réécriture de multi-ensembles. Une molécule peut hiérarchiquement contenir d'autres solutions contenues dans des membranes perméables aux valences. La machine la plus simple est le crible de Darwin de la figure 11 pour calculer les nombres premiers. Il utilise la règle chimique p, kp $\rightarrow$ p (tout nombre mange ses multiples). Un exemple plus complexe lié à CCS est présenté dans les transparents et vidéos.

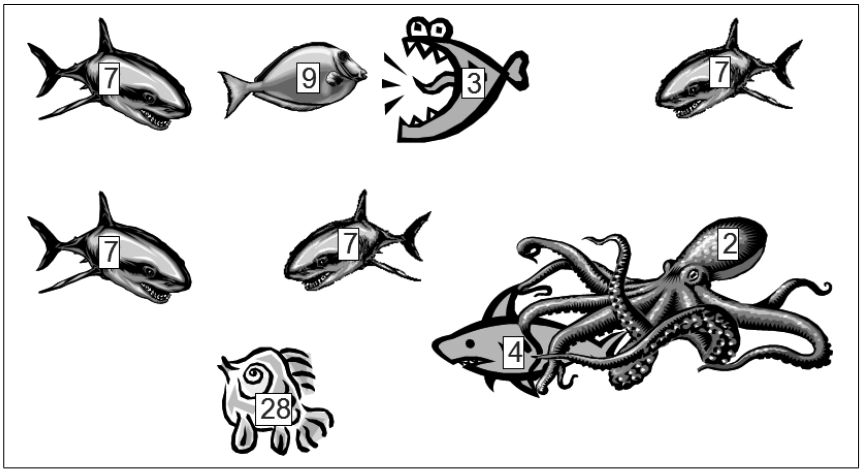

Figure 11 : **le crible chimique de Darwin**

La machine chimique est plus simple et plus intuitive que les formalismes précédemment utilisés pour les processus parallèles, par exemple la présentation logique SOS (*Structural Operational Semantics*) de Plotkin, bien plus puissante mais pas indispensable ici.

4.6. Le π-calcul

Après avoir étudié les calculs de processus asynchrones de type CCS, qui ont fait faire des progrès considérables à la théorie du parallélisme, Milner a établi une jonction fondamentale entre le λ-calcul et les processus asynchrones en définissant le *π-calcul* [Sangiorgi et Walker, 2003]. Ce calcul se compose de processus asynchrones qui communiquent par des échanges de noms sur des canaux eux-mêmes nommés par ces noms. Le processus $a<b>.P$ envoie le nom b sur le canal a et se continue en Q, alors que le processus a(x).P reçoit un nom qu'il lie à la variable x dans Q comme en λ-calcul, puis se continue après substitution en Q[b/x]. La mise en parallèle de deux processus P et Q est notée P | Q. La communication est régie par la règle $a<b>.P \mid a(x).Q \rightarrow P \mid Q[b/x]$, inspirée par la β-conversion du λ-calcul. Une primitive (υx) P permet de créer un nouveau nom inconnu jusqu'alors et de le lier à x dans P, rendant le calcul complètement dynamique. La grande différence avec le λ-calcul est qu'on se restreint à passer des noms atomiques au lieu des termes complets, ce qui est indispensable en univers parallèle distribué. Cependant, Milner a montré que cette restriction n'est pas essentielle, car le π-calcul peut simuler le λ-calcul.

Le π-calcul est particulièrement expressif car il permet d'exprimer des calculs dynamiques avec migration des données et processus. Mais il est mathématiquement complexe dans sa version initiale. Il a été simplifié par plusieurs auteurs. Boudol

et Honda en ont indépendamment donné une version chimique vraiment asynchrone équivalente mais plus simple. Dans d'autres versions, des restrictions sur la portée des variables ont supprimé des interférences non souhaitables sans changer l'expressivité (Lπ-calcul de Sangiorgi *et. al.*). Des calculs plus orientés vers la programmation ont été développés, comme le *Join-Calcul* de Fournet, Gonthier et Lévy et les *ambients* de Cardelli. Enfin, le π-calcul a de nouvelles applications très importantes et non prévues à l'origine, comme la description et l'analyse formelles des réactions chimiques dans la cellule en biologie ou la définition et la preuve de protocoles de sécurité informatique, sujet qui fera l'objet du cours 2010-2011 de Martin Abadi sur la chaire Informatique et sciences numériques.

4.7. Séminaires sur le parallélisme asynchrone

Le parallélisme asynchrone a fait l'objet de deux séminaires. Le premier, de Michel Raynal, professeur à l'université de Rennes, a porté sur les grands algorithmes distribués comme l'élection d'un leader ou la recherche d'un consensus [Raynal, 1992]. Le second, de Cédric Fournet, a porté sur le *Join-Calcul* mentionné ci-dessus et son implémentation dans le langage JoCaml, une extension parallèle et surtout distribuée de Caml.

5. Parallélisme synchrone

Le parallélisme synchrone avait été déjà décrit dans le cours 4 de 2008, « systèmes embarqués ». Il est à la base des langages de programmations Esterel, Lustre et SCADE, utilisés industriellement pour la conception de circuits et de systèmes temps-réels critiques. Le cours 2009 en a présenté les principes généraux, la puissance d'expression et la sémantique mathématique. Il a en particulier discuté les fondements théoriques du langage Esterel et du formalisme graphique associé SyncCharts, leur sémantique mathématique, et leur traduction en circuits électroniques efficaces. Nous ne présentons pas ces sujets ici faute de place. Mais le développement d'Esterel nous a également conduits à des résultats nouveaux sur la traduction des automates en circuits, que nous présentons ici car ils complètent ceux de la section 3. Des résultats analogues ont été trouvés indépendamment par Pascal Raymond, du laboratoire Verimag à Grenoble.

5.1. Compilation d'expressions rationnelles en circuits

Revenons à l'automate déterministe construit efficacement par l'algorithme de Berry Sethi en 3.5. Nous avons vu que cet automate a pour taille maximale le carré du nombre de lettres. On peut l'implémenter en circuit électronique en le dessinant directement de la façon suivante :

– chaque état est représenté par un registre, initialisé à 1 pour l'état initial, à 0 sinon, et alimenté par une porte « ou » ;

– chaque transition part du registre de son état de départ, passe par une porte « et » à laquelle est aussi connectée l'entrée de la lettre concernée, puis rejoint la porte « ou » connectée à l'entrée de son état d'arrivée ;

– la sortie est constituée d'une porte « ou » alimentée par les entrées des états.

On vérifie sans peine que la sortie du circuit vaut 1 si et seulement si le mot d'entrée (présenté une lettre par cycle) est reconnu par l'automate. Au cours de l'exécution, l'ensemble des registres peut prendre exactement les valeurs considérées dans l'algorithme de déterminisation présenté en 3.6. En fait, le circuit réalise la déterminisation « au vol » de l'automate non-déterministe, *se comportant comme l'automate déterministe tout en évitant totalement l'explosion exponentielle de la déterminisation*. Pour une compilation matérielle, il est facile de fabriquer un code C de taille linéaire dans la taille du circuit en ordonnant les équations en fonction des dépendances de variable (ce qu'on appelle un simulateur « cycle-based » du circuit). L'explosion en taille est donc aussi évitée en logiciel.

Une extension de cette technique fondamentale est utilisée pour le compilateur industriel Esterel v7, qui compile le langage Esterel et les SyncCharts. Cette méthode et d'autres sont présentées en [Potop-Botucaru, Edwards et Berry, 2008].

5.2. Caractérisation constructive des circuits cycliques synchrones

La compilation d'Esterel en circuits a mis à jour un phénomène insoupçonné, la génération de circuits dont la partie combinatoire est cyclique. Faute de place, nous n'en étudierons pas ici la raison (voir les vidéos et transparents pour un exemple de protocole symétrique d'accès à un bus ; bien d'autres exemples naturels sont apparus dans notre coopération avec Dassault Aviation sur des spécifications avioniques, en particulier d'interfaces homme-machine). Dans un autre registre, Sharad Malik, de Princeton, a montré que les circuits cycliques peuvent dans certain cas être exponentiellement plus petits que les circuits acycliques pour produire un comportement donné, ce qui peut aussi être intéressant pratiquement.

Mais les circuits cycliques sont rejetés par tous les systèmes de CAO de circuits synchrones, car leur comportement peut être indéfini. En effet, des équations comme « X = X » et « X = non X » ne définissent pas un comportement booléen synchrone, puisque la première a deux solutions et la seconde aucune. Si on les réalise par des circuits électroniques, la première fournit un voltage indéterminé et la seconde une oscillation perpétuelle. Mais ce n'est pas le cas pour tous les circuits cycliques. Considérons par exemple le circuit C suivant, d'entrées I et J et de sorties X et Y :

– X = Y et I ;

– Y = X et J.

Alors I = 0 entraîne X = 0 puis Y = 0 quel que soit Y, ceci à la fois en booléen et en électronique, quels que soient les délais des fils et portes du circuit. Symétriquement, J = 0 entraîne Y = 0 puis X = 0. Mais, si I = J = 1, le circuit se réduit à X = Y, Y = X, indéterminé. On voit donc qu'un circuit cyclique peut être bien déterminé ou non selon ses entrées.

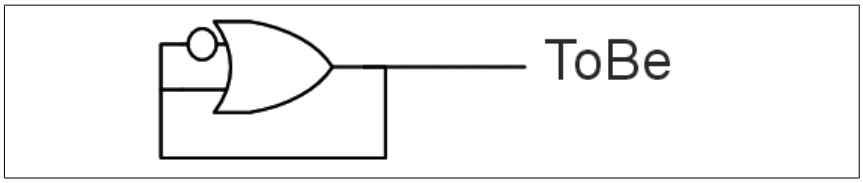

Figure 12 : **Hamlet : ToBe = ToBe or not ToBe**

Une question se pose immédiatement : étant donné un circuit cyclique, peut on caractériser les entrées pour lesquelles il a un comportement électronique synchrone, stabilisant électriquement ses sorties sur les valeurs uniques solutions de ses équations booléennes ? Il faut alors diviser les circuits cycliques en trois catégories : les mauvais comme $X = X$, les sans-problèmes comme C ci-dessus si $I = 0$ ou $J = 0$, et les douteux, dont l'amusant prototype est Hamlet : ToBe = ToBe or not ToBe. Vu d'un logicien classique, on a ToBe = 1 par tiers exclu. Mais, électriquement, il existe des délais des fils et portes pour lesquels le circuit oscille sans se stabiliser. La logique classique ne s'applique donc pas.

La solution est de remplacer la logique classique par la *logique constructive*, qui refuse le tiers exclu, et n'admet de déduire ToBe or not ToBe = 1 que si on a auparavant déduit ToBe = 0 ou ToBe = 1 ; l'information ne peut pas y être déduite de rien. Le théorème central, conjecturé par moi-même puis prouvé par Shiple et Mendler, résout le problème général : un circuit se stabilise pour tous délais étant donnée une entrée si et seulement si ses sorties peuvent être calculées seulement en utilisant la logique constructive. Il fournit aussi un algorithme pour calculer pour quelles entrées le circuit se stabilise toujours, et, au besoin, d'en donner une version équivalente acyclique.

Ce beau résultat non-trivial relie le modèle physique, quantifié sur des délais réels, et le modèle logique dans sa version constructive. Il est à rapprocher de la correspondance de Curry-Howard entre λ-calcul et logique intuitionniste (cf. 2.4), mais avec ici un caractère plus opérationnel.

5.3. Séminaires sur le parallélisme synchrone

Le parallélisme synchrone a fait l'objet de deux séminaires. Le premier, de Nicolas Halbwachs, directeur de recherches CNRS à Grenoble, a porté sur le noyau du langage Lustre, base de SCADE. Le second, de Jean Vuillemin, professeur à l'École normale supérieure, a porté sur l'application du modèle synchrone aux circuits électroniques et sur le magnifique modèle 2-adique qu'il a défini pour ces circuits.

Lors de ma répétition du cours en anglais à Édimbourg, un concert sera organisé par Arshia Cont, de l'IRCAM. La musique est évidemment une instance naturelle du parallélisme synchrone. Le concert présentera *Anthèmes II* de Pierre Boulez joué

par une violoniste virtuose, et accompagné automatiquement par ordinateur à partir de partitions décrites dans un langage synchrone de type Esterel. Grâce à un remarquable système d'analyse prédictive des événements musicaux en temps réel, ces partitions sont déclenchées automatiquement par un système qui analyse le jeu de l'interprète et s'adapte à son tempo, même s'il est irrégulier.

6. Le parallélisme diffus (séminaires)

Les applications modernes en réseau introduisent un nouveau type de parallélisme, qu'on peut appeler parallélisme diffus. Ici, des acteurs potentiellement en grand nombre peuvent apparaître et disparaître, coopérant ou entrant en compétition de façon temporellement limitée, tombant en panne, etc. De plus, les informations échangées par ces acteurs peuvent être approximatives plutôt qu'exactes. Par exemple, quand on transmet un film sur internet en continu (streaming), il est toujours préférable de perdre des images plutôt que de perdre le fil du temps si la qualité de la communication se dégrade. Demander la retransmission d'une image dont le temps est passé serait idiot. L'information n'est plus exacte, mais orientée vers une certaine « satisfaction » du receveur qu'il est rarement facile de définir.

Le sujet a été abordé sous la forme de trois séminaires. Le premier, par Manuel Serrano, directeur de recherches à l'INRIA Sophia-Antipolis, a présenté le langage HOP dédié à la programmation compréhensible d'applications distribuées sur le Web. Étendant HTML par une base fonctionnelle SCHEME, ce langage permet de programmer de façon centralisée des applications distribuées, puis de répartir automatiquement le code sur les agents concernés. Deux applications pratiques ont illustré son expressivité et sa puissance. Le second séminaire, par Anne-Marie Kermarrec, directrice de recherches à l'INRIA à Rennes, a montré comment utiliser de nouvelles techniques de communication comme le *gossiping* (ragotage) pour traiter de façon nouvelle les requêtes d'un moteur de recherches, en posant la question seulement à la communauté concernée. La difficulté est évidemment de détecter automatiquement cette communauté, ce qui demande des algorithmes probabilistes exploitant la structure naturelle de graphe « petit monde » qu'on trouve sur les liens dans le Web. Le dernier séminaire, par Sophie Denève, chercheuse à l'École normale supérieure, s'est intéressé aux calculs probabilistes faits par le cerveau pour résoudre des problèmes et a montré la robustesse des réseaux probabilistes pour des actions de reconnaissance.

Le séminaire de Manuel Serrano sera redonné dans la traduction du cours à Edimbourg en octobre 2010.

7. Coopération de modèles de calcul

Les modèles de calculs précités ne sont évidemment pas exclusifs. De fait, les applications modernes demandent souvent la coopération de plusieurs d'entre eux. Par exemple, les circuits électroniques modernes de type « systèmes sur puces » sont composés de nombreux composants localement synchrones (processeurs, accélérateurs divers, caméras, gestionnaires mémoires, etc.) reliés entre eux de façon

globalement asynchrone par des bus ou des réseaux complexes sur puce. Sur un plan logiciel, on trouve des structures analogues dans les grands simulateurs (avions, réseau électrique, etc.) ou dans les jeux en réseau. Le dernier cours a présenté les difficultés associées à la composition de modèles, qui reste encore mal maîtrisée et est un sujet actif de recherche. Il a montré par exemple l'importance du problème de la métastabilité des registres matériels, qui exprime qu'on ne peut pas transmettre de façon fiable une information d'une zone d'horloge à une autre dans un circuit sans utiliser un protocole complexe, à cause du fait qu'un registre devient métastable (non 0/1) si son entrée est en train de changer quand arrive le front de l'horloge qui le gouverne.

7.1. Séminaire de neuro-informatique

Ce cours a été accompagné d'un séminaire de neuro-informatique donné par Yves Frégnac, directeur de recherches CNRS à Gif-sur-Yvette, qui a montré comment des mécanismes de calcul dans le cortex visuel provoquent des formes de calcul « cristallines », « liquides », ou « en fumée », à partir d'informations codées sous forme de trains d'impulsions (*spikes*), et pourquoi ces formes de calcul sont intrinsèquement robustes à la modification et à la variabilité du substrat neuronal.

Ce séminaire sera redonné en septembre 2010 à Edimbourg pour la version anglaise du cours.

Conclusion

Notre tour des modèles formels du calcul automatique nous a permis de comprendre la vision moderne du calcul automatique, très riche et parfois surprenante. Ce domaine est en pleine activité productive, avec de grands problèmes ouverts comme la compréhension et la maîtrise des processus asynchrones, de leur programmation et de leur vérification. Les électroniciens nous promettent des circuits multicœurs avec des centaines ou des milliers de processeurs. Saurons-nous en tirer parti ? La question est loin d'être claire, et la programmation des ordinateurs et réseaux de l'avenir demandera certainement la conjonction de toutes les méthodes présentées ici de façon individuelle.

Bibliographie

[Allouche et Shallit, 2003] : J.-P. Allouche et J. Shallit, *Automatic Sequences: Theory, Applications, Generalizations,* Cambridge University Press, 2003.

[Amadio et Curien, 1998] : R. Amadio et P.-L. Curien, *Domains and Lambda-Calculi*, Cambridge University Press, 1998.

[Barendregt, 1987] : H. Barendregt, *The Lambda-Calculus, its Syntax and Semantics*, Elsevier Science Ltd, 1987.

[Berry et Curien, 1985] : G. Berry et P.-L. Curien, « Theory and Practice of Sequential Algorithms: the Kernel of the Programming Language CDS », dans *Algebraic Methods in Semantics*, Cambridge University Press, 1985, 35-88.

[Berry et Boudol, 1992] : G. Berry et G. Boudol, « The Chemical Abstract Machine », *Theoretical Computer Science*, vol. 96, 1992, 217-248.

[Berry, 2008] : G. Berry, *Pourquoi et comment le monde devient numérique*, Fayard-Collège de France, 2008.

[Berry, 2009] : G. Berry, *Penser, modéliser et maîtriser le calcul informatique*, Fayard-Collège de France, 2010.

[Berry et Sethi, 2009] : G. Berry et R.Sethi, « From Regular Expressions to Deterministic Automata », *Theoretical Computer Science*, 48, 1986, 117-126.

[Gonthier, 2008] : G. Gonthier, « Formal Proof – The Four-Color Theorem », *Notices of the American Mathematical Society*, 55(11), 2008, 1382-1393.

[Herlihy et Shavit, 2009] : M. Herlihy et N. Shavit, *The Art of Multiprocessor Programming.*, Morgan Kaufmann, 2008.

[Landin, 1966] : P.-J. Landin, « The next 700 Programming Languages », *Communications of the ACM*, vol. 9, nº 3, 1966, 157-166.

[McQueen, 2009] : D. McQueen, « Kahn Networks at the Dawn of Functional Programming », *From Semantics to Computer Science*, Morgan-Kaufmann, 2009.

[Milner, 1980] : R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag, 1980.

[Potop-Botucaru, E. et G. Berry, 2008] : Dumitru Potop-Butucaru, Stephen Edwards et Gérard Berry, *Compiling Esterel*, Springer, 2008.

[Raynal, 1992] : M. Raynal, *Synchronisation dans les systèmes répartis*, Eyrolles, 1992.

[Razet, 2009] : B. Razet, *Machines d'Eilenberg effectives*, thèse d'informatique, université Paris VII, 2009.

[Sakarovitch, 2003] : J. Sakarovitch, *Éléments de théorie des automates*, Vuibert Informatique, 2003.

[Sangiorgi et Walker, 2003] : D. Sangiorgi, D. Walker, *The Pi-Calculus: A Theory of Mobile Processes*, Cambridge University Press, 2003.

Autres activités liées au cours

Traduction en anglais

Avec celui de 2008, le cours sera partiellement repris dans sept leçons en en anglais à l'université d'Edimbourg, en partenariat avec la Royal Society of Edinburgh, le titre global étant « Seven Keys to the Digital Future ». Ces leçons et les séminaires associés seront disponibles sur Internet.

Conférences associées aux cours

Conférence « Pourquoi et comment le monde devient numérique », association Sapience, Nice, 7 novembre 2009.

Participation à une table ronde au colloque « Mathématiques et industrie », Paris, 1er décembre 2009.

Conférence « Science informatique et informatisation des sciences », Congrès STIC de l'ANR (Agence nationale de la recherche), Paris, 5 janvier 2010.

Conférence « Pourquoi et comment le monde et les sciences deviennent numériques » au Visiatome, Marcoule, 11 février 2010.

Conférence sur le monde numérique dans le cadre « L'informatique, entre pratique et savoirs », Maison des sciences humaines et sociales (MESHS), Lille, 11 mars 2010.

Conférence « Qualité du logiciel : certification *vs.* vérification », journées nationales du GDR génie logiciel, Pau, 12 mars 2010.

Conférence « Les évolutions techniques et mentales qui impactent la DSI », club OLG, Paris, 16 mars 2010.

Conférence « L'évolution du monde numérique » à l'Institut Aspen, Maison des polytechniciens, 1er avril 2010.

Conférence « Pourquoi et comment le monde et les sciences deviennent numériques », Assemblée générale des ingénieurs Supélec, 8 avril 2010.

Conférence « Pourquoi le numérique bouleverse les comportements, la culture et les sciences », Editis, congrès des éditeurs de manuel scolaires, 3 juin 2010.

Conférence « Thinking About, Modeling, and Mastering Computation », congrès Frontiers of Neuromorphic Computing, Collège de France, 3 juin 2010.

Conférence « Du temps réel au temps logique », dans le cycle Modèle/Prototype/Œuvre, IRCAM, 10 juin 2010.

Conférence « La révolution numérique en marche et l'évolution des usages », Entretiens d'Opio des DSI (directeurs de services informatiques), 11 juin 2010.

Conférence « La révolution numérique en marche et l'évolution des usages », Direction des services informatiques) Angers, 25 juin 2010.

Conférence « L'informatique en biologie et médecine : d'un auxiliaire de calcul à une nouvelle façon de penser », projet BioIntelligence, 5 juillet 2010.

Conférence invitée « What could be the right balance between abstract and fine-grain computational properties ? » au workshop LOLA (Low-Level Languages), Edimbourg, Ecosse, 9 juillet 2010.

Conférence invitée « How many fundamental parallel computation models ? » au workshop DCM (Designing Computation Models), Edimbourg, Ecosse, 9 juillet 2010.

Actions sur l'enseignement

J'ai continué l'action engagée en 2008 pour l'introduction de l'enseignement d'informatique au lycée. La proposition initiale d'option scientifique en seconde ayant été finalement supprimée avec l'annulation de la réforme Darcos, le ministère de l'Éducation nationale a proposé la création d'un enseignement de spécialité informatique en terminale en 2012. J'ai participé à un groupe de travail sur la formation des professeurs de lycée, en liaison avec l'inspection générale. J'ai continué mes conférences dans les lycées ou pour les professeurs initiées en 2008 :

– Conférence « Pourquoi et comment le monde et les sciences deviennent numériques », Fête de la science, Centre International de Valbonne, 16 novembre 2009.

– Participation à une table ronde sur l'enseignement de l'informatique au salon *Éducatice*, Paris, 20 novembre 2009.

– Participation aux rencontres de l'Orme sur l'enseignement numérique, Marseille, 31 mars 2010.

– Conférence « Pourquoi et comment le monde devient numérique », Lycée Audiberti, Antibes, 6 mai 2010.

– Conférence « Pourquoi et comment enseigner la science informatique (et pas seulement ses usages) », Journée du numérique, Université de Paris Descartes, 17 mai 2010.

– Conférence « Pourquoi et comment le monde devient numérique » aux lycéens et professeurs de seconde, INRIA Nancy, 7 juin 2010.

– Visio-conférence « Pourquoi l'école devient numérique » aux professeurs de l'Académie de Montpellier, 6 juillet 2010.
– Conférences en classes de seconde et de mathématiques spéciales au Lycée Alphonse Daudet de Nîmes, 22 septembre 2010.

Retombées médiatiques

J'ai participé aux émissions radiophoniques suivantes :
Radio libre de Ali Baddou sur France Culture, le 20 novembre 2009.
La tête au Carré de Mathieu Vidard sur France Inter, les 19 janvier et 8 février 2010.
Science publique de Michel Alberganti sur France Culture, le 14 mai 2010.

J'ai également enregistré un CD sur le monde numérique fondé sur mon cours, pour la collection « De vive voix », en partenariat avec l'Académie des sciences.

Articles de journaux

J'ai écrit des articles de vulgarisation pour les journaux suivants : *TDC* (textes et documents pour la classe), *La Recherche*, *La jaune et la rouge*. Des interviews ou des articles sur la chaire sont parus dans l'*Ordinateur individuel*, *Maths à venir*, *Le Monde*, *Le Point*, *Les Échos*, *La Recherche*, la revue *Pluriels* de l'association APM progrès management, et le *Journal du CNRS*.