UFR mathématiques et informatique **École doctorale IAEM Lorraine**

Département de formation doctorale en informatique

Analyse de la complexité et transformation de programmes

THÈSE

présentée et soutenue publiquement le 17 décembre 2003

pour l'obtention du

Doctorat de l'université Nancy 2

(spécialité informatique)

par

Jean-Yves Moyen

Composition du jury

Rapporteurs :	Roberto Amadio, Professeur Neil D. Jones, Professeur
Examinateurs :	Bruno Gaujal, Directeur de recherches Claude Kirchner, Directeur de recherches Jean-Yves Marion, Professeur (directeur) Jeanine Souquières, Professeur (présidente)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Je tiens tout d'abord à remercier mon directeur de thèse, Jean-Yves Marion. Au cours de ces presque quatre années de travail en commun il a su me diriger et m'encourager pour m'apprendre le métier de chercheur et je lui dois beaucoup.

Je remercie aussi vivement Neil D. Jones qui a toujours regardé ce que je faisais et a très gentiment accepté de m'accueillir à Copenhague ce qui a permis de grandes avancées dans mon travail. Je suis très content qu'il ait accepté d'être rapporteur de ma thèse.

Je remercie aussi Roberto Amadio d'avoir accepté d'être rapporteur et Bruno Gaujal, Claude Kirchner et Jeanine Souquières d'avoir accepté de faire partie du jury.

Les discussions et explications de Bruno Gaujal sur les réseaux de Petri ont été cruciales pour mon travail. De même, certaines discussions plus ou moins informelles sur les graphes avec Frédéric Mazoit, Hervé Rivano et Bernard Helmstetter m'ont grandement aidé. Merci à eux tous.

Ma mère a patiemment relu, parfois dans l'urgence, les versions préliminaires du manuscrit. Je la remercie pour les nombreuses corrections qu'elle m'a permis de faire.

Depuis toujours mes parents et ma famille m'ont éduqué et soutenu. Il est évident que je ne serais pas là sans eux et je les remercie de tout ce qu'ils ont fait pour moi. Rémi a su me supporter quotidiennement pendant de longues années et en particulier pendant ma thèse, je lui en sais gré. Jean-François et Myriam m'ont fait une charmante petite nièce toute souriante et j'espère qu'il y en aura d'autres.

Les musiciens et danseurs de la Taverne de l'Irlandais m'ont permis de décompresser de temps en temps en pensant à autre chose qu'à ma thèse. Je remercie plus particulièrement le trio Carnyx, Anne-Claire, Laurent, Caro, Tom, Em, Coucou, Kelly, Seb, Gallou, Francis, Seb, ...et j'espère que ceux que j'aurais oublié voudront bien me pardonner.

Merci aussi à Philippe Thibaut d'avoir osé écrire *Europa Universalis*. Merci à Risto Marjomaa, Pierre Borgnat et Bertrand Asseray d'avoir retravaillé ce jeu et merci à Rémi, Aurélien, Benjamin, Éric, Jean-Christophe et Sylvain de m'avoir permis d'en faire une partie en entier.

Merci à Aymeric Gillet, Gwenaël Samotyj et Laurent Mousson d'avoir créé ATPUB. Merci, notamment, à messieurs Lancelot, Boon, Van Roy et aux moines des abbayes d'Orval et de Westvleteren. Merci à Virgile Perrin pour son approvisionnement sans faille, à Jean-François Simard et Mario D'Eer pour leurs ouvrages et à Jean-Claude Cunillera pour son matériel.

Merci, en vrac, à MaD, Vince, Claire, Vincent, Cabud, Juliette, Jolow, RHNK, Benj, Hyperion, Babasse, Kysly, VV, Kloecky, lapin, Glo, DL, LCQ, Zebio, Rémy, Alfred, Alvin, Fabien, Seb, Titi, Olivier, Émelyne, JC, Bilbo, Anne-Claire, Caroline, Laurent, Troll et tous les autres pour leur amitié pendant tout ce temps.

Table des matières

Introduction

Calculabilité et complexité

Depuis la mise en place de la théorie de la calculabilité dans les années 1930 par A. Turing [Tur36], A. Church [Chu38] et S. C. Kleene [Kle52] (entre autres), on sait que certaines fonctions ne sont pas calculables. Cependant, même parmi les fonctions calculables (c'est-à-dire qu'on peut écrire un programme, dans n'importe quel langage de programmation, qui calcule cette fonction), il en existe qui sont plus simples que d'autres. C'est à partir de cette constatation que se met en place la théorie de la complexité.

La théorie de la complexité permet de dire si une fonction donnée est calculable avec une certaine borne sur les ressources (par exemple le temps ou l'espace de calcul). Elle était sous-jacente à la théorie de la calculabilité dès le début et a connu un essor important avec l'apparition des premiers ordinateurs, puis dans les années 1970 avec les travaux de S. Cook [Coo70] et W. J. Savitch [Sav70] notamment.

Cependant, une dissymétrie fondamentale est présente dans la théorie de la complexité. Il est possible de donner une borne supérieure sur la complexité d'une fonction (*eg* combien faut-il de temps ou d'espace pour être sûr de pouvoir calculer telle fonction) mais il est extrêmement difficile de donner, de manière effective, une borne inférieure (*eg* quelle quantité de temps ou d'espace est insuffisante pour calculer telle fonction). En effet, on ne peut étudier la complexité des fonctions que par l'étude de la complexité des algorithmes les calculant, mais il est presque impossible de déterminer si un algorithme est optimal. De plus, le simple calcul de la complexité d'un algorithme est très difficile.

À partir de là, deux approches coexistent. D'une part, on peut essayer de donner des restrictions à priori sur la forme des algorithmes permettant ainsi de borner la complexité des algorithmes écrits. Cette approche a été introduite par A. Cobham [Cob62] puis reprise par S. Bellantoni et S. Cook [BC92], D. Leivant et J.-Y. Marion [LM93], avec l'introduction de l'analyse prédicative, puis N. D. Jones, H. Schwichtenberg, M. Hofmann et d'autres. On peut ainsi obtenir des familles d'algorithmes qui permettent de calculer toutes les fonctions d'une classe de complexité et uniquement celles-ci.

Cette approche doit être faite avec précaution. En effet, il ne sert à rien de caractériser une classe de complexité par un ensemble d'algorithmes si ces algorithmes ne sont jamais utilisés en pratique. De même, si un algorithme largement répandu pour calculer une fonction donnée n'apparaît pas dans la caractérisation, on peut se dire qu'on a un peu manqué son but et que la caractérisation n'est pas aussi utile qu'elle pourrait l'être. Cet aspect très important de la caractérisation des classes de complexité est ce qu'on appelle l'*intentionnalité*. Une caractérisation serait intentionnellement complète si elle contenait tous les algorithmes utilisés en pratique pour calculer les fonctions de la classe voulue. Cette complétude intentionnelle est, hélas, inatteignable en pratique car, par exemple, l'ensemble des algorithmes calculant en temps polynômial n'est pas récursif et probablement pas récursivement énumérable. Il faut donc se contenter de l'approcher autant que possible et essayer sans cesse d'avoir des caractérisations aussi complètes intentionnellement que possible.

D'autre part, on peut aussi chercher à savoir quel est le lien entre la complexité de l'algorithme observé et celle de la fonction calculée. Ainsi, on devrait pouvoir obtenir des indications sur la complexité des fonctions. Cet aspect est celui de la *complexité implicite*. On ne cherche plus ici à obtenir la complexité de la fonction à partir de la complexité de l'algorithme, mais on cherche à classer les algorithmes selon la complexité des fonctions calculées (ce qui est loin d'être évident car il existe plusieurs algorithmes pour calculer chaque fonction). Là aussi, cette approche demande un certain recul et une certaine réflexion. En effet, il est peu utile de connaître la complexité d'une fonction calculée par un programme si cette connaissance n'est pas constructive et ne permet pas d'exhiber un algorithme qui a effectivement la complexité annoncée.

Il devient alors nécessaire de transformer le programme en un autre programme, qui calcule la même fonction mais dont la complexité est connue grâce à l'analyse préalable. Le premier exemple de transformation de programmes est le théorème *s-m-n* de S. C. Kleene [Kle52] qui implique l'existence de spécialiseurs et de l'évaluation partielle [JGS93] ; viennent ensuite des techniques comme la déforestation de P. Wadler [Wad90] ou la supercompilation de V. Turchin [Tur86] et l'automatisation de techniques algorithmiques de programmation, comme la programmation dynamique qui donne naissance à la mémoïsation de N. D. Jones d'après des travaux de S. Cook [Coo71, AJ94] puis à l'incrémentalisation de A. Liu et D. S. Stoller [LS99].

Analyse et transformation

L'analyse de programmes, au sens large, peut alors se subdiviser en deux catégories. D'une part l'analyse proprement dite qui permet de prouver certaines propriétés d'un programme comme par exemple la terminaison, une borne sur la complexité ou le fait de calculer en espace constant. Et d'autre part la transformation de programmes qui a pour but d'obtenir, à partir d'un programme initial, un autre programme calculant la même fonction mais de manière "plus efficace". Bien sûr, il est extrêmement intéressant de mélanger les deux approches. On pourrait alors obtenir un programme transformé pour lequel certaines propriétés ont été prouvées par la transformation elle-même.

L'analyse de programmes a des applications évidentes dans tous les domaines où on fait une confiance importante au système informatique. Par exemple, il est utile de connaître les propriétés des programmes embarqués dans les voitures, métros, avions ou autres afin d'éviter certains accidents. De même, connaître les propriétés des codes exécutés sur des machines distantes (par exemple du code Java reçu par internet ou transmis par une carte à puce à un automate) peut grandement améliorer la sécurité de l'application.

La transformation de programmes, elle, a plutôt des applications du côté du développement de logiciels. En effet, on constate que les programmes écrits par des êtres humains, et facilement compréhensibles, sont généralement peu efficaces alors que des programmes optimisés pour l'ordinateur sont généralement peu compréhensibles pour des êtres humains. On peut par exemple comparer l'écriture récursive (et exponentielle) de la fonction de Fibonacci avec l'écriture itérative (linéaire) de cette même fonction, ou alors comparer un programme écrit en `C` avec le code assembleur produit par un compilateur qui effectue beaucoup d'optimisations. Ainsi, on pourrait imaginer un environnement de programmation intentionnelle, c'est-à-dire permettant aux programmeurs de développer un logiciel dans un langage de spécifications de très haut niveau, et donc facilement débugable, mais l'exécutant après lui avoir fait subir des transformations (plus poussées que la seule compilation) afin d'avoir toujours un programme efficace.

Contributions de la thèse

Cette thèse s'articule autour de deux grandes parties et présente deux résultats principaux concernant l'analyse d'une part des programmes fonctionnels du premier ordre et d'autre part d'un langage assembleur.

Les systèmes de réécriture permettent de simuler facilement les langages fonctionnels du premier ordre. Ils ont été beaucoup analysés, notamment par le biais d'ordres de terminaison. D. Hofbauer, A. Cichon ou A. Weirmann ont montré que les systèmes de réécriture terminant par l'ordre MPO (resp. LPO) caractérisent les classes de complexité PrimRec (resp. MultRec). Mais ces caractérisations sont peu utiles pour le programmeur tant les classes concernées sont grandes.

Je montre qu'on peut restreindre ces ordres de terminaison par l'introduction de quasi-interprétations polynomiales. On obtient alors un effet similaire à celui de l'analyse prédicative et je montre ainsi que les systèmes de réécriture terminant par MPO (resp. LPO) et admettant une quasi-interprétation polynomiale caractérisent la classe de complexité Ptime (resp. Pspace). En fait, on peut même aller plus loin et classer les systèmes selon la forme des quasi interprétations. En reprenant la classification des polynômes en genre établie dans [BCMT00], on obtient le tableau suivant indiquant la classe caractérisée en fonction de l'ordre de terminaison.

Genre de la Quasi-interprétation	Système confluent		Système non confluent	
	MPO	*LPO*	*MPO*	*LPO*
0	Ptime	Pspace	NPtime	(N)Pspace
1	E_1time	E_1space	NE_1time	$(N)E_1$space
2	E_2time	E_2space	NE_2time	$(N)E_2$space

De plus, j'ai écrit un petit programme baptisé ICAR qui permet de tester si un système de réécriture entre dans cette caractérisation.

La deuxième partie de la thèse concerne l'analyse de programmes écrits dans un langage assembleur simplifié. Les programmes sont d'abord transformés en réseaux de Petri et ces réseaux sont ensuite analysés. J'ai tout d'abord développé cette technique pour essayer d'étendre le champ d'application du Size-Change Principle [LJBA01], c'est-à-dire pour pouvoir détecter la terminaison d'un plus grand nombre de programmes. De manière fort intéressante, cette technique permet, outre la terminaison, d'analyser d'autres propriétés du programme comme le calcul en place et même d'effectuer simplement des transformations de programme élémentaires assimilables à la déforestation. Je pense qu'elle peut être étendue pour englober la plupart des techniques actuelles d'analyse ou de transformation de programmes donnant ainsi naissance à une sorte d'outil générique.

Plan de la thèse

Le premier chapitre décrit les différents modèles de calculs utilisés par la suite, c'est-à-dire principalement les systèmes de réécriture et les machines à compteurs. J'introduis aussi les machines de Turing à cet endroit, bien que je ne les utilise pas ailleurs, car elles servent généralement de base théorique à la calculabilité et permettent donc de montrer que les deux modèles utilisés peuvent calculer les mêmes fonctions.

Le deuxième chapitre définit les diverses classes de complexité utilisées ainsi que les relations d'inclusions qui existent entre elles.

Le troisième chapitre présente les résultats déjà existants sur l'analyse de terminaison des systèmes de réécriture. Les ordres de terminaison sont présentés, notamment MPO et LPO, ainsi que les interprétations polynomiales et les résultats qui vont avec.

Le quatrième et le cinquième chapitre présentent les résultats obtenus au cours de ma thèse sur les systèmes de réécriture. Les ordres de terminaison du chapitre précédent sont d'abord restreints par l'analyse prédicative, puis les quasi-interprétations polynomiales sont introduites et les résultats correspondants sont démontrés.

Le sixième chapitre présente rapidement ce que sont les réseaux de Petri et démontre quelques lemmes utiles pour leur analyse ainsi que quelques aspects algébriques de l'analyse des réseaux de Petri et des graphes.

Le septième chapitre montre comment les programmes assembleurs, représentés par des machines à compteurs, peuvent être modélisés par des réseaux de Petri. Je construis ainsi un "Size-Change Petri Net" qui permet de suivre précisément l'évolution de la taille des variables lors de l'exécution d'un programme assembleur et dont la terminaison implique celle du programme.

Le huitième chapitre ajoute à ces réseaux une gestion de la mémoire libre, permettant ainsi de caractériser les programmes calculant "en place" (*ie* pouvant être écrits sans `malloc`).

Le neuvième chapitre regroupe toutes les techniques nécessaires à l'analyse des réseaux de Petri, en particulier pour vérifier la terminaison d'un réseau sous certaines hypothèses.

Chapitre 1

Modèles de calculs

Ce chapitre présente trois modèles de calculs utilisés par la suite et établit leur équivalence. Le premier modèle est celui des machines de Turing. Il est utilisé depuis longtemps comme modèle théorique des calculs et base des théories de la calculabilité et de la complexité. Sa présence ici permet de raccrocher les deux autres modèles à cette base théorique et de s'assurer ainsi qu'ils ont une puissance de calcul suffisante.

Si on reprend la définition que donne N. D. Jones au chapitre 6 de son livre *Computability and Complexity from a Programming Perspective* [Jon97], un langage de programmation se définit par :

- Deux ensembles de programmes et de données.
- Un fonction de sémantique qui prend un programme et une donnée et renvoie une donnée qui est le résultat du calcul.
- Une fonction de mesure du temps qui prend un programme et une donnée et renvoie le temps nécessaire au calcul du programme sur cette donnée.

Ces deux ensembles et deux fonctions ne sont pas définis explicitement pour chacun des trois modèles de calculs présentés ici, mais il est facile de voir que cette définition ne poserait aucun problème.

1.1 Machines de Turing

1.1.1 Syntaxe

Les machines de Turing sont un modèle de calculs qui a été établi par A. Turing dans les années 1930 [Tur36, TG95]. Depuis, elles ont été largement étudiées et servent maintenant de référence pour définir ce qui est calculable par un ordinateur classique. De nombreuses variations sur le modèle initial existent (en faisant varier le nombre ou la topologie des rubans, la taille de l'alphabet, ...) qui sont toutes équivalentes en terme de fonctions calculables.

Ici, le choix a été fait d'utiliser des machines à deux rubans, un pour les entrées uniquement (donc sur lequel on ne peut pas écrire) et un ruban de travail, sur lequel on peut lire et écrire.

Informellement, une machine de Turing comporte 2 rubans bi-infinis. Sur chacun d'eux est positionnée une tête de lecture. En fonction de la lettre lue sur chaque ruban et de l'état de la machine, un automate permet de passer dans un nouvel état, de déplacer les têtes de lecture et d'écrire une nouvelle lettre **sur le deuxième ruban**. Initialement, les données sont inscrites sur une partie du premier ruban. Le reste du premier ruban et le deuxième ruban sont blancs. Quand la machine atteint un certain état, on peut lire le résultat du calcul sur le deuxième ruban.

Le choix d'une machine à deux rubans (au lieu d'un seul ou d'un nombre quelconque) permet d'une part d'avoir un modèle relativement simple à décrire (par rapport aux modèles avec un nombre de rubans quelconque) et d'autre part de faire abstraction du premier ruban (en lecture seule) dans le calcul de l'espace utilisé, ce qui permettra d'avoir des complexités en espace sous-linéaire, et notamment la classe LOGSPACE.

Les machines sont présentées ici avec une notion de description instantanée et un pas de calcul similaire à la réduction des systèmes de réécriture. Ceci permet de définir une normalisation des descriptions instantanées et de donner une approche similaire de tous les modèles de calculs présentés.

Définition 1 (Machine de Turing).
Une *machine de Turing* M est un quintuplet (Σ, Q, q_0, q_f, F) où :
- Σ est un *alphabet* fini comportant au moins 2 symboles dont le symbole spécial $\sqcup$ qui représente le blanc.
- Q est un ensemble fini d'*états*.
- $q_0 \in Q$ est l'*état initial*.
- $q_f \in Q$ est l'*état final*.
- $F : Q \times \Sigma \times \Sigma \to Q \times \Delta \times \Delta \times \Sigma$ est la *fonction de transition* avec $\Delta = \{\to, \downarrow, \leftarrow\}$.

1.1.2 Sémantique

Définition 2 (Mots).
Soit Σ un alphabet (*ie* un ensemble de lettres). Un *mot* sur Σ est une suite finie de lettre $\omega = a_1 \cdots a_n$. Le mot vide est noté ϵ.
On note Σ^* l'ensemble des mots sur Σ.

Pour représenter chaque ruban bi-infini d'une machine de Turing, on utilise deux piles. Cela correspond à couper le ruban en un endroit donné (qui sera celui de la tête de lecture) et représenter par une pile la zone qui comporte tous les symboles différents de $\sqcup$. Pour faciliter les déplacements de la tête de lecture, il faut que les lettres au sommet de chaque pile soient adjacentes sur le ruban (le déplacement de la tête correspond alors à déplacer une lettre d'une pile à l'autre). Il faut donc "renverser" la partie gauche du ruban dans la pile qui la représente. Chaque pile est ensuite simplement représentée par un mot.

Définition 3 (Pas de calcul).
Soit M une machine de Turing. Une *description instantanée* de M est un quintuplet $p = (\omega_1, \omega_2, \nu_1, \nu_2, q)$ où :
- $\omega_1, \omega_2, \nu_1, \nu_2$ sont quatre mots finis. ω_1 et ω_2 forment le premier *ruban*, ν_1 et ν_2 forment le deuxième *ruban*. Les premières lettres de ω_1 et ν_1 sont dites *marquées*. Elles désignent l'emplacement, sur chaque ruban, des *têtes de lecture*.
- $q \in Q$ est un état.

Un *pas de calcul* permet de passer d'une description instantanée à une autre. Si $\omega_1 = a_i \dots a_1$, $\omega_2 = a_{i+1} \dots a_n$, $\nu_1 = b_j \dots b_1$ et $\nu_2 = b_{j+1} \dots b_p$ (si l'un des quatre mots est vide, on considère qu'il est constitué de plusieurs $\sqcup$), alors un pas de calcul permet de passer de $p_1 = (\omega_1, \omega_2, \nu_1, \nu_2, q)$ à $p_2 = (\omega_1', \omega_2', \nu_1', \nu_2', q')$ avec :
- $F(q, a_i, b_j) = (q', \delta_1, \delta_2, c)$.
- Si $\delta_1 = \downarrow$ alors $\omega_k' = \omega_k$ (la tête 1 ne bouge pas).
- Si $\delta_1 = \to$ alors $\omega_1' = a_{i+1} a_i \dots a_1$ et $\omega_2' = a_{i+2} \dots a_n$ (la tête 1 va vers la droite).
- Si $\delta_1 = \leftarrow$ alors $\omega_1' = a_{i-1} \dots a_1$ et $\omega_2' = a_i a_{i+1} \dots a_n$ (la tête 1 va vers la gauche).
- Si $\delta_2 = \downarrow$ alors $\nu_1' = c b_{j-1} \dots b_1$ et $\nu_2' = \nu_2$ (la tête 2 ne bouge pas).
- Si $\delta_2 = \to$ alors $\nu_1' = b_{j+1} c b_{j-1} \dots b_1$ et $\nu_2' = b_{j+2} \dots b_p$ (la tête 2 va vers la droite).
- Si $\delta_2 = \leftarrow$ alors $\nu_1' = b_{j-1} \dots b_1$ et $\nu_2' = c b_{j+1} \dots b_p$ (la tête 2 va vers la gauche).

c'est-à-dire que la fonction de transition prend en entrée la lettre lue par chacune des deux têtes de lecture et l'état courant et renvoie la lettre à écrire sur le deuxième ruban, le nouvel état courant et le déplacement à effectuer par chaque tête de lecture.

On note $M \vdash p_1 \to p_2$.

Définition 4 (Normalisation).
On note $\xrightarrow{+}$ la clôture transitive de $\to$, c'est-à-dire que $M \vdash p_0 \xrightarrow{+} p_n$ si il existe des positions $p_1, \dots, p_{n-1}$ telles que pour tout i, $M \vdash p_i \to p_{i+1}$.
On note $\xrightarrow{*}$ la clôture réflexive et transitive de $\to$. c'est-à-dire que $M \vdash p \xrightarrow{*} p'$ si $p = p'$ ou $M \vdash p \xrightarrow{+} p'$.
Si l'état de la position atteinte est l'état final, on dit qu'on a *normalisé* la position initiale. On note $M \vdash p \xrightarrow{!} p'$ si $p' = (\omega_1, \omega_2, \nu_1, \nu_2, q_f)$.

Soit M une machine de Turing. On dit que M calcule la fonction $f : \Sigma^* \to \Sigma^*$ si en partant de l'état initial, avec le mot écrit sur le premier ruban et la tête de lecture positionnée au début de celui-ci, on obtient après normalisation le résultat écrit sur le deuxième ruban.

Définition 5 (Calcul).
Formellement, M calcule une fonction partielle f si pour tout mot fini ω tel que $f(\omega) = \nu$ on a

$$M \vdash (\omega, \epsilon, \epsilon, \epsilon q_0) \xrightarrow{!} (\omega_1', \omega_2', \nu', \nu, q_f)$$

On note alors $f = [M]$.

Exemple 1.
La machine de Turing suivante prend en entrée un mot écrit avec des **0** et des **1** et vérifie si il s'agit d'un palindrome. Elle commence par recopier le mot sur le deuxième ruban (q_0), puis elle parcours les deux mots, un dans chaque sens, en vérifiant que les lettres sont identiques (q_2). Si le résultat est un palindrome elle termine en écrivant **1** au début du deuxième mot (*ie* au début du résultat), sinon elle écrit **0** au début du résultat.

- $\Sigma = \{\mathbf{0}, \mathbf{1}, \sqcup\}$.
- $Q = \{q_0, \ldots, q_2, q_f\}$.
- F est décrite par le tableau ci-dessous, x désigne n'importe quelle lettre et $\mathbf{i}, \mathbf{j}$ désignent **0** ou **1**.

État	Lettre$_1$	Lettre$_2$	État	Mvt$_1$	Mvt$_2$	Écrire
q_0	$\mathbf{i}$	x	q_0	$\rightarrow$	$\rightarrow$	$\mathbf{i}$
q_0	$\sqcup$	x	q_1	$\leftarrow$	$\leftarrow$	x
q_1	$\mathbf{i}$	x	q_1	$\leftarrow$	$\downarrow$	x
q_1	$\sqcup$	x	q_2	$\rightarrow$	$\downarrow$	x
q_2	$\mathbf{i}$	$\mathbf{i}$	q_2	$\rightarrow$	$\leftarrow$	$\mathbf{i}$
q_2	$\sqcup$	$\sqcup$	q_f	$\downarrow$	$\downarrow$	$\mathbf{1}$
q_2	$\mathbf{i}$	$\mathbf{j} \neq \mathbf{i}$	q_f	$\downarrow$	$\downarrow$	$\mathbf{0}$

1.1.3 Complexité

Définition 6 (Temps de calcul).
Soient M une machine de Turing et ω un mot fini. Le *temps de calcul* de M sur ω est égal au nombre de pas de calcul nécessaires pour calculer $[M](\omega)$, c'est-à-dire au nombre de pas de calcul nécessaires pour normaliser $(\omega, \epsilon, \epsilon, \epsilon, q_0)$.

Définition 7 (Taille des mots).
Soit ω un mot. Sa *taille* $|\omega|$ est égale à son nombre de lettres.

Définition 8 (Complexité en temps).
Soit M une machine de Turing. La *complexité en temps* de M est la plus petite fonction T telle que pour tout mot fini ω, le temps de calcul de M sur ω est inférieur ou égal à $T(|\omega|)$.

Cette définition n'est pas très formelle dans la mesure où il peut être délicat de comparer deux fonctions. L'idée généralement admise est de faire une comparaison asymptotique, c'est-à-dire que f est "plus petite" que g si au voisinage de l'infini on a $g = O(f)$. Ceci ne donne toujours pas l'unicité de la complexité (il peut y avoir plusieurs fonctions qui sont minimales pour cet ordre). On prend alors la fonction "la plus simple" comme complexité.

On peut cependant remarquer que la définition précise de *la* complexité en temps d'une machine de Turing n'est pas très importante, ce qui est important c'est sa capacité à calculer en temps polynômial (ou exponentiel ou autre) qui mettra la fonction calculée dans telle ou telle classe de complexité. Les équivalences entre fonctions n'ont alors plus grande importance.

Comme le note Y. Gurevich [Gur93], la comparaison asymptotique n'est pas forcément une bonne idée dans la mesure ou une fonction "plus grande" qu'une autre à l'infini peut en fait très bien être plus petite et avec une croissance plus lente pour les valeurs sur lesquelles le calcul est susceptible d'avoir lieu. Par exemple, n^{100}, bien qu'étant un polynôme, croît beaucoup plus vite que $n^{log(n)}$ pour des valeurs de n raisonnables (inférieures à $2^{100} \approx 10^{30}$).

Exemple 2.
La machine de Turing pour reconnaître les palindromes a un temps de calcul maximum égal à 3 fois la longueur du mot (car le mot est parcouru une fois par q_0 pour le recopier sur le deuxième ruban, une deuxième fois par q_1 pour revenir au début et une troisième fois par q_2 pour comparer).

Sa complexité en temps est donc $n \mapsto 3 \times n$. De manière plus globale, cette complexité est linéaire.

Définition 9 (Espace de calcul).
Soient M une machine de Turing et ω un mot fini. L'*espace de calcul* de M sur ω est le maximum des tailles des mots écrits sur le deuxième ruban pendant la normalisation de $(\omega, \epsilon, \epsilon, \epsilon, q_0)$.

C'est-à-dire le maximum pour chaque description instantanée $(\omega_1, \omega_2, \nu_1, \nu_2, q)$ des sommes des tailles de ν_1 et ν_2.

Définition 10 (Complexité en espace).
Soit M une machine de Turing. La *complexité en espace* de M est la plus petite fonction E telle que pour tout mot fini ω, l'espace de calcul de M sur ω est inférieur ou égal à $E(|\omega|)$.

La même remarque que pour la complexité en temps s'applique ici en ce qui concerne la comparaison des fonctions.

Exemple 3.
La complexité en espace de la machine présentée précédemment est $n \mapsto n$. En effet, elle utilise exactement autant de place sur le deuxième ruban que la taille de l'entrée.

1.2 Systèmes de réécriture

1.2.1 Syntaxe

Les systèmes de réécriture présentés ici permettent de modéliser les langages fonctionnels du premier ordre. La plupart des notations et définitions sont reprises de l'étude de N. Dershowitz et J.-P. Jouannaud [DJ90]. On pourra aussi consulter l'ouvrage de C. et H. Kirchner [KK99].

Définition 11 (Syntaxe).
Soient $\mathcal{X}, \mathcal{F}, \mathcal{C}$ trois ensembles distincts de variables, symboles de fonctions et symboles de constructeurs. Les ensembles de termes, motifs et règles sont définis par la grammaire suivante :

$$
\begin{array}{llcl}
\texttt{(Termes constructeurs)} & \mathcal{T}(\mathcal{C}) \ni u & ::= & \mathbf{c}(u_1, \ldots, u_n) \\
\texttt{(Termes clos)} & \mathcal{T}(\mathcal{C}, \mathcal{F}) \ni s & ::= & \mathbf{c}(s_1, \ldots, s_n) \mid \mathtt{f}(s_1, \ldots, s_n) \\
\texttt{(Termes)} & \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t & ::= & x \mid \mathbf{c}(t_1, \ldots, t_n) \mid \mathtt{f}(t_1, \ldots, t_n) \\
\texttt{(Motifs)} & \mathcal{P} \ni p & ::= & \mathbf{c}(p_1, \ldots, p_n) \mid x \\
\texttt{(Règles)} & \mathcal{D} \ni d & ::= & \mathtt{f}(p_1, \cdots, p_n) \to t
\end{array}
$$

Par la suite, les symboles de fonctions seront systématiquement écrits en fonte «`machine à écrire`» et les symboles de constructeurs en **gras**.

Définition 12 (Programmes).
Un programme est un quadruplet $main = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ tel que :
- $\mathcal{E}$ est un ensemble de règles.
- Chaque variable apparaissant dans le côté droit d'une règle apparaît aussi dans le côté gauche de la même règle.
- Il existe un symbole de fonction spécial appelé symbole principal qui est noté `main`.

Typage des programmes

Définition 13 (Types).
Soit $\mathcal{A} = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ un ensemble de types atomiques. L'ensemble des *types* construits à partir de $\mathcal{A}$ est *Types* $\ni \tau ::= \mathtt{a} \to \tau | \mathtt{a}$.

On peut remarquer que chaque type est de la forme $\mathtt{a}_1 \to (\cdots(\mathtt{a}_n \to \mathtt{a})\cdots)$ et représente donc une fonction. On écrira plus simplement $(\mathtt{a}_1, \ldots, \mathtt{a}_n) \to \mathtt{a}$.

Définition 14 (Environnement de typage).
Un *environnement de typage* est une fonction Γ de $\mathcal{X} \bigcup \mathcal{C} \bigcup \mathcal{F}$ vers *Types* qui assigne à chaque symbole

(de constructeur ou de fonction) f d'arité $n > 0$ un type $\Gamma(f) = (\mathtt{a}_1, \ldots, \mathtt{a}_n) \rightarrow \mathtt{a}$, et à chaque symbole d'arité 0 un type atomique.

Définition 15 (Typage correct).
Soit Γ un environnement de typage. Un terme $t \in \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$ est de type τ, noté $t : \tau$, si et seulement si :
- t est un symbole d'arité 0 et $\Gamma(t) = \tau$.
- $t = f(t_1, \ldots, t_n)$, $t_i : \mathtt{a}_i$ et $\Gamma(f) = \tau = (\mathtt{a}_1, \ldots, \mathtt{a}_n) \rightarrow \mathtt{a}$.

Définition 16 (Programme typé).
Un *programme typé* est un sextuplet $main = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \mathcal{A}, \Gamma \rangle$ tel que $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ est un programme (non typé), $\mathcal{A}$ un ensemble de types et Γ un environnement de typage et pour chaque règle $\mathtt{f}(p_1, \ldots, p_n) \rightarrow t$, les types de $\mathtt{f}(p_1, \ldots, p_n)$ et t sont les mêmes vis-à-vis de Γ.

1.2.2 Sémantique

Définition 17 (Sous-terme).
Soient t et s deux termes. On dit que t est un *sous-terme* de s, et on note $t \trianglelefteq s$ si :
- $t = s$;
- ou $s = f(s_1, \ldots, s_n)$ $(f \in \mathcal{F} \bigcup \mathcal{C})$ et il existe i tel que $t \trianglelefteq s_i$.

Lemme 1.
$\trianglelefteq$ est une relation d'ordre sur $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$.

Démonstration.
Elle est réflexive, anti-symétrique et transitive. □

Définition 18 (Substitution).
Une *substitution* σ est une fonction de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$. Une *substitution close* est une fonction de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{C}, \mathcal{F})$. Une *substitution-constructeur* est une fonction de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{C})$.

Soit σ une substitution, on note $\sigma\{x \leftarrow t\}$ la substitution qui associe t à x et $\sigma(y)$ à chaque autre variable.

Les substitutions sont étendues canoniquement aux termes de la manière suivante :
- $\sigma(\mathtt{f}(t_1, \ldots, t_n)) = \mathtt{f}(\sigma(t_1), \ldots, \sigma(t_n))$.
- $\sigma(\mathbf{c}(t_1, \ldots, t_n)) = \mathbf{c}(\sigma(t_1), \ldots, \sigma(t_n))$.

Afin d'économiser les parenthèses, on note parfois $t\sigma$ au lieu de $\sigma(t)$.

Définition 19 (Rédex).
Soit *main* un programme et t un terme clos. Un *rédex* de t est un terme u tel que :
- $u \trianglelefteq t$.
- Il existe une règle $d = \mathtt{f}(p_1, \ldots, p_n) \rightarrow v$ et une substitution σ telles que $\sigma(\mathtt{f}(p_1, \ldots, p_n)) = u$.

On peut alors *réduire* t en s, obtenu en remplaçant u par $\sigma(v)$ dans t. On note $t \rightarrow s$.

Définition 20 (Réduction, normalisation).
La relation $\xrightarrow{+}$ est la clôture transitive de $\rightarrow$.
La relation $\xrightarrow{*}$ est la clôture transitive et relative de $\rightarrow$, c'est-à-dire que $t\xrightarrow{*}s$ si $t = s$ ou $t\xrightarrow{+}s$.
Un terme qui ne contient aucun rédex, et ne peut donc pas être réduit, est dit en *forme normale*. La relation $\xrightarrow{!}$ est la relation de *normalisation*. c'est-à-dire que $t\xrightarrow{!}s$ si $t\xrightarrow{*}s$ et s est en forme normale.

En toute généralité, la forme normale d'un terme donné n'est pas unique. Comme je cherche ici à représenter des fonctions, je veux un seul résultat pour chaque entrée. En outre, quelques restrictions supplémentaires sur les systèmes de réécriture vont permettre de représenter assez fidèlement les langages fonctionnels du premier ordre (comme une restriction de Caml ou SML au premier ordre).

Définition 21 (Confluence).
Un système de réécriture est *confluent* si deux termes ayant un ancêtre commun ont forcément un descendant commun, c'est-à-dire que si il existe t, s et u tels que $t\xrightarrow{*}s$ et $t\xrightarrow{*}u$, alors il existe v tel que $s\xrightarrow{*}v$ et $u\xrightarrow{*}v$.

Lemme 2.
Si un système de réécriture est confluent alors chaque terme a au plus une forme normale.

Démonstration.
Si il existe un terme t qui a deux formes normales s et u, alors par confluence il existe un terme v tel que $s \xrightarrow{*} v$ et $u \xrightarrow{*} v$. Mais s et u sont des formes normales et ne peuvent donc pas se réduire. On a donc forcément $s = v = u$ et la forme normale est unique. $\square$

Par la suite et sauf mention explicite du contraire, on considère uniquement des systèmes de réécriture confluents. G. Huet [Hue80] donne divers critères pour déterminer si un système de réécriture est confluent.

Définition 22.
La sémantique d'un type τ est l'ensemble des termes constructeurs de type τ.
$[\tau] = \{t : \tau \mid t \in \mathcal{T}(\mathcal{C})\}$.

La fonction calculée par un programme est celle obtenue en normalisant les termes. Toutefois, seules les formes normales constructeurs sont considérées comme correctes.

Définition 23 (Sémantique).
Soient *main* un programme typé et $\Gamma(\mathtt{main}) = (\tau_1, \ldots, \tau_n) \to \tau$. La *fonction calculée* par *main* est $[\mathtt{main}] : [\tau_1] \times \ldots \times [\tau_n] \mapsto [\tau]$ définie comme suit :

Pour chaque $u_i \in [\tau_i]$, $[\mathtt{main}](u_1, \ldots, u_n) = v$ si et seulement si $\mathtt{main}(u_1, \ldots, u_n) \xrightarrow{!} v$ et $v \in [\tau]$. Dans tous les autres cas, $[\mathtt{main}](u_1, \ldots, u_n)$ n'est pas définie.

Exemple 4.
Le système de réécriture suivant trie une liste selon l'algorithme du tri par insertion : à chaque étape, un élément est inséré dans la liste à la bonne position. Les types sont $\{\mathtt{Bool}, \mathtt{Nat}, \mathtt{List}\}$, les constructeurs (et leur type) sont $\{\mathbf{True} : \mathtt{Bool}, \mathbf{False} : \mathtt{Bool}, \mathbf{0} : \mathtt{Nat}, \mathbf{S} : \mathtt{Nat} \to \mathtt{Nat}, \mathbf{Nil} : \mathtt{List}, \mathbf{Cons} : \mathtt{Nat} \times \mathtt{List} \to \mathtt{List}\}$. Les types des fonctions sont donnés avec les règles de réécritures correspondantes. Le symbole de fonction $\leq$ a été infixé pour plus de lisibilité.

$\_ \leq \_ : \mathtt{Nat} \times \mathtt{Nat} \to \mathtt{Bool}$.

$$\begin{aligned} \mathbf{0} \leq y &\to \mathbf{True} \\ \mathbf{S}(x) \leq \mathbf{0} &\to \mathbf{False} \\ \mathbf{S}(x) \leq \mathbf{S}(y) &\to x \leq y \end{aligned}$$

$\mathtt{insert} : \mathtt{Nat} \times \mathtt{List} \to \mathtt{List}$.

$$\begin{aligned} \mathtt{insert}(x, \mathbf{Nil}) &\to \mathbf{Cons}(x, \mathbf{Nil}) \\ \mathtt{insert}(x, \mathbf{Cons}(y, \mathbf{l})) &\to \mathtt{insert'}(x \leq y, x, y, \mathbf{l}) \end{aligned}$$

$\mathtt{insert'} : \mathtt{Bool} \times \mathtt{Nat} \times \mathtt{Nat} \times \mathtt{List} \to \mathtt{List}$.

$$\begin{aligned} \mathtt{insert'}(\mathbf{True}, x, y, \mathbf{l}) &\to \mathbf{Cons}(x, \mathbf{Cons}(y, \mathbf{l})) \\ \mathtt{insert'}(\mathbf{False}, x, y, \mathbf{l}) &\to \mathbf{Cons}(y, \mathtt{insert}(x, \mathbf{l})) \end{aligned}$$

$\mathtt{main} = \mathtt{sort} : \mathtt{List} \to \mathtt{List}$.

$$\begin{aligned} \mathtt{sort}(\mathbf{Nil}) &\to \mathbf{Nil} \\ \mathtt{sort}(\mathbf{Cons}(x, \mathbf{l})) &\to \mathtt{insert}(x, \mathtt{sort}(\mathbf{l})) \end{aligned}$$

1.2.3 Complexité

Définition 24 (Temps de calcul).
Soient *main* un programme et $t_1, \ldots, t_n$ des termes constructeurs. Le *temps de calcul* de *main* sur $t_1, \ldots, t_n$ est égal au nombre (minimal) de réductions nécessaires pour calculer $[\![\mathtt{main}]\!](t_1, \ldots, t_n)$, c'est à dire pour normaliser $\mathtt{main}(t_1, \ldots, t_n)$.

Cette définition considère que chaque pas de réduction prend un temps constant. Ceci appelle deux remarques :

- Tout d'abord, on peut noter que si on devait implémenter un système de réécriture, chaque réduction ne prendrait pas un temps constant. En effet, pour trouver le rédex il faut parcourir tout le terme et le temps nécessaire à chaque réduction serait donc proportionnel à la taille du terme à réduire. Toutefois, sur un calcul donné, on peut borner ce surcoût par une constante multiplicative. Comme on s'en rendra aisément compte au chapitre suivant, cette constante ne change en rien la définition des classes de complexité et en conséquence ne pose aucun problème pour la suite.
- Ensuite, la définition proposée ici est, justement, une *définition* de la complexité pour un modèle de calculs donné et absolument pas une *mesure* du temps de calculs que mettrait un vrai ordinateur pour simuler ce modèle. Le fait que la simulation d'un modèle de calculs par un autre (ici des systèmes de réécriture par les ordinateurs) soit moins efficace que l'original est bien connu et sera développé plus amplement à la fin de ce chapitre. En conséquence, il n'y a aucun problème à définir la complexité des systèmes de réécriture à partir du seul nombre de réductions et sans tenir compte des problèmes pratiques que pourraient causer ces réductions sur un ordinateur.

Définition 25 (Taille des termes).
Soit t un terme. Sa taille est définie récursivement de la manière suivante :

- $|x| = 1$.
- $|\mathtt{f}(s_1, \ldots, s_n)| = 1 + \sum_{i=1}^{n} |s_i|$.
- $|\mathbf{c}(s_1, \ldots, s_n)| = 1 + \sum_{i=1}^{n} |s_i|$.

Comme on l'aura sans doute remarqué, certaines notations sont communes aux différents modèles de calculs, comme par exemple la taille, la normalisation, la relation d'étape de calcul élémentaire (pas de calcul ou réduction), ... Ceci est volontaire et permet de montrer que ces notions sont similaires dans chaque modèle, même si elles n'ont pas exactement la même définition à chaque fois. En pratique, le contexte permettra de déterminer dans quel modèle de calculs on se place et donc de quelle notion et définition il s'agit.

Définition 26 (Complexité en temps).
Soit *main* un système de réécriture. La *complexité en temps* de *main* est la plus petite fonction T telle que pour tous termes $t_1, \ldots, t_n$, le temps de calcul de *main* sur $t_1, \ldots, t_n$ est inférieur ou égal à $T(\sum_{i=1}^{n} |t_i|)$.

La remarque concernant la complexité en temps des machines de Turing est toujours valable ici.

Définition 27 (Espace de calcul).
Soient *main* un système de réécriture et $t_1, \ldots, t_n$ des termes constructeurs. L'*espace de calcul* d'une chaîne de réductions est le maximum des tailles des termes intermédiaires nécessaires au calcul de $[\![\mathtt{main}]\!](t_1, \ldots, t_n)$, c'est à dire à la normalisation de $\mathtt{main}(t_1, \ldots, t_n)$ le long de cette chaîne de réduction.

L'*espace de calcul* de *main* sur $t_1, \ldots, t_n$ est le minimum des espaces de calcul le long de chaque chaîne de réduction partant de $\mathtt{main}(t_1, \ldots, t_n)$.

Exemple 5.
Soit le système de réécriture suivant :

$\mathtt{d} : \mathtt{Nat} \to \mathtt{Nat}.$

$$\mathtt{d}(\mathbf{0}) \to \mathbf{0}$$
$$\mathtt{d}(\mathbf{S}(x)) \to \mathbf{S}(\mathbf{S}(\mathtt{d}(x)))$$

$\mathtt{exp} : \mathtt{Nat} \to \mathtt{Nat}$.

$$\begin{aligned} \mathtt{exp}(\mathbf{0}) &\to \mathbf{S}(\mathbf{0}) \\ \mathtt{exp}(\mathbf{S}(x)) &\to \mathtt{d}(\mathtt{exp}(x)) \end{aligned}$$

$\mathtt{f} : \mathtt{Nat} \to \mathtt{Nat}$.

$$\mathtt{f}(x) \to \mathbf{0}$$

$\mathtt{main} : \mathtt{Nat} \to \mathtt{Nat}$.

$$\mathtt{main}(x) \to \mathtt{f}(\mathtt{exp}(x))$$

Le terme $\mathtt{main}(\mathbf{S}(\mathbf{S}(\mathbf{0})))$ admet plusieurs chaînes de réduction. L'une d'elle est

$$\mathtt{main}(\mathbf{S}(\mathbf{S}(\mathbf{0}))) \to \mathtt{f}(\mathtt{exp}(\mathbf{S}(\mathbf{S}(\mathbf{0})))) \to \mathbf{0}$$

Une autre est

$$\mathtt{main}(\mathbf{S}(\mathbf{S}(\mathbf{0}))) \to \mathtt{f}(\mathtt{exp}(\mathbf{S}(\mathbf{S}(\mathbf{0})))) \xrightarrow{*} \mathtt{f}(\mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{0}))))) \to \mathbf{0}$$

La première chaîne de réduction a un espace de calcul de 5 alors que la deuxième a un espace de calcul de 6. Si on considère l'espace de calcul de $\mathtt{main}(\mathbf{S}(\mathbf{S}(\mathbf{0})))$, on prend le *minimum* des espaces de calcul de chaque chaîne de réduction et on obtient donc 4.

Définition 28 (Complexité en espace).
Soit *main* un système de réécriture. La *complexité en espace* de *main* est la plus petite fonction E telle que pour tous termes $t_1, \dots, t_n$, l'espace de calcul de *main* sur $t_1, \dots, t_n$ est inférieur ou égal à $E(\sum_{i=1}^{n} |t_i|)$.

Là encore, les fonctions sont comparées asymptotiquement.

Contrairement aux machines de Turing, la complexité en espace d'un système de réécriture est au moins linéaire. En effet, la simple écriture du terme initial force $E(N) \geq N + 1$. Pour les machines de Turing, le fait de mesurer l'espace de calcul uniquement sur le deuxième ruban (initialement vide) permet de se passer de cette limite et d'avoir des algorithmes qui calculent en espace sous-linéaire (typiquement, en espace logarithmique).

1.3 Machines à compteurs

1.3.1 Syntaxe

Les machines à compteurs ont été introduites par J. C. Shepherdson et H. E. Sturgis [SS63]. On présente ici un mini langage machine qui permet aisément de les simuler pas à pas. On dispose de plusieurs compteurs, qui sont des entiers, et d'un jeu réduit d'instructions qui permettent d'incrémenter ou décrémenter les compteurs ou de se rendre à un endroit précis du programme. Par rapport aux langages machines des processeurs actuels, les machines à compteurs ont deux manques principaux. D'une part l'absence d'opérations arithmétiques (mais elles sont assez facilement implémentables) et d'autre part l'absence de stockage externe (par exemple une pile ou un tas). En revanche, elles disposent d'un nombre quelconque de compteurs alors que les microprocesseurs n'ont qu'un nombre fini de registres de calcul. Formellement, le langage est défini par la grammaire suivante :

Définition 29 (Machines à compteur).

$$\begin{array}{rcl} \mathtt{pgm} & ::= & \mathtt{entrée} : i_1, \dots, i_m ; \mathtt{sortie} : o_1, \dots, o_n ; \\ & & 0 : \mathtt{inst}_0 ; \dots ; n : \mathtt{inst}_n ; end \\ \mathtt{inst} & ::= & \mathtt{dec}\ x \mid \mathtt{inc}\ x \mid \mathtt{jmp}\ \mathtt{lbl}_i \mid \mathtt{jz}\ x\ \mathtt{lbl}_i \mid \\ & & \mathtt{call}\ \mathtt{lbl}_i \mid \mathtt{return} \\ x & ::= & \text{nom de variable} \\ \mathtt{lbl} & ::= & 0 | \dots | n | end \end{array}$$

Les $i_1, \ldots, i_m$ sont les *variables d'entrée* et les $o_1, \ldots, o_n$ sont les *variables de sortie*. Une variable peut être à la fois variable d'entrée et de sortie. x représente n'importe quelle variable, qu'elle soit d'entrée, de sortie, ou encore une nouvelle variable utilisée uniquement à l'intérieur du programme. Chaque variable est un *compteur* et stocke une valeur entière positive. Un programme calcule ainsi une fonction de $\mathbb{N}^m$ vers $\mathbb{N}^n$.

L'instruction `inc` (resp. `dec`) incrémente (resp. décrémente) une variable. L'instruction `jmp` fait directement sauter l'exécution du programme au label indiqué. L'instruction `jz` saute si et seulement si la variable indiquée vaut 0. Les instructions `call` et `return` permettent de faire un appel à une sous routine et un retour à l'endroit de l'appel.

Définition 30.
On définit une fonction `instr` qui prend comme argument un label et renvoie l'instruction qui s'y trouve. c'est-à-dire que $\texttt{instr}(i) = \texttt{inst}_i$.

Le mécanisme des sous-routines permet de factoriser facilement du code qui est utilisé plusieurs fois. Cependant si on le mélange sans faire attention avec des `jmp`, on risque vite d'arriver à un véritable sac de nœuds totalement inextricable. Afin d'éviter ça, on impose une règle de bon sens aux sous-routines, à savoir que chaque sous-routine doit former un tout dont on ne peut sortir que par l'unique `return`.

Définition 31 (Sous-routines).
L'ensemble $\Xi(\texttt{lbl})$ des labels *atteignables* depuis un label `lbl` est le plus petit ensemble qui vérifie :

1. $\texttt{lbl} \in \Xi(\texttt{lbl})$.
2. Si $\texttt{lbl}' \in \Xi(\texttt{lbl})$ et $\texttt{instr}(\texttt{lbl}') \in \{\texttt{inc}\ x, \texttt{dec}\ x, \texttt{call}\ \texttt{lbl}'', \texttt{jz}\ x\ \texttt{lbl}''\}$, alors $\texttt{lbl}' + 1 \in \Xi(\texttt{lbl})$.
3. Si $\texttt{lbl}' \in \Xi(\texttt{lbl})$ et $\texttt{instr}(\texttt{lbl}') \in \{\texttt{jmp}\ \texttt{lbl}'', \texttt{jz}\ x\ \texttt{lbl}''\}$, alors $\texttt{lbl}'' \in \Xi(\texttt{lbl})$.

Une *sous-routine* est l'ensemble des labels atteignables depuis un label `lbl` tel qu'il existe une instruction `call lbl` dans le programme.

Définition 32 (Correction syntaxique).
Un programme est *syntaxiquement correct* si :

- Chaque label appartient à au plus une sous-routine.
- Chaque sous-routine contient une et une seule instruction `return`.

Par la suite, on s'intéresse uniquement aux programmes syntaxiquement corrects.

1.3.2 Sémantique

Les programmes sont interprétés par la sémantique d'appels par valeurs relativement standard présentée à la figure 1.1.

Comme pour les systèmes de réécriture, les substitutions associent des variables à des valeurs, ici des entiers naturels uniquement.

Définition 33 (Substitution).
Une *substitution* σ est une fonction des variables vers les entiers naturels. La substitution $\sigma\{x \leftarrow v\}$ est celle qui assigne v à x et toutes les autres variables à la même valeurs que σ. De même, $\{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_m\}$ est la substitution qui assigne les v_i aux x_i et 0 à chaque autre variable.

Définition 34 (Pile de retour).
Une *pile de retour* `stk` est une liste de labels. La pile vide est notée []. $\texttt{lbl} :: l$ est la pile dont `lbl` est la tête et l la queue.

Définition 35 (État).
Un *état du programme* est soit un triplet $(\texttt{lbl}, \sigma, \texttt{stk})$ où `lbl` est un label, σ est une substitution et `stk` une pile de retour, soit l'état spécial $\perp$ utilisé pour gérer les erreurs.

Définition 36 (Sémantique).
La relation $\rightarrow$ telle que définie dans la figure 1.1 correspond à l'exécution d'une instruction du programme. Comme pour les modèles précédents, $\xrightarrow{+}$ est la clôture réflexive de $\rightarrow$ et $\xrightarrow{*}$ est la clôture réflexive et

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{inc}\ x}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \to \mathtt{lbl}+1, \sigma\{x \leftarrow \sigma(x)+1\}, \mathtt{stk}}\ \text{incrément}$$

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{dec}\ x \quad \sigma(x) = 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \to \bot}\ \text{erreur}$$

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{dec}\ x \quad \sigma(x) \neq 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \to \mathtt{lbl}+1, \sigma\{x \leftarrow \sigma(x)-1\}, \mathtt{stk}}\ \text{décrément}$$

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{jmp}\ \mathtt{lbl}'}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \to \mathtt{lbl}', \sigma, \mathtt{stk}}\ \text{saut}$$

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{jz}\ x\ \mathtt{lbl}' \quad \sigma(x) = 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \to \mathtt{lbl}', \sigma, \mathtt{stk}}\ x = 0$$

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{jz}\ x\ \mathtt{lbl}' \quad \sigma(x) \neq 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \to \mathtt{lbl}+1, \sigma, \mathtt{stk}}\ x \neq 0$$

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{call}\ \mathtt{lbl}'}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \to \mathtt{lbl}', \sigma, (\mathtt{lbl}+1, \mathtt{stk})}\ \text{appel}$$

$$\frac{\mathtt{instr}(\mathtt{lbl}) = \mathtt{return}}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, [\mathtt{lbl}', \mathtt{stk}] \to \mathtt{lbl}', \sigma, \mathtt{stk}}\ \text{retour}$$

FIG. 1.1 – Sémantique d'appels par valeurs.

transitive de $\rightarrow$. $\xrightarrow{!}$ correspond à la normalisation, c'est-à-dire que $s\xrightarrow{!}s'$ si et seulement si $s\xrightarrow{*}s'$ et $s' = end, \sigma, []$.

Le programme pgm calcule la fonction $[\texttt{pgm}]$ définie par $[\texttt{pgm}](v_1, \ldots, v_n) = (\sigma(o_1), \ldots, \sigma(o_n))$ si $\texttt{pgm} \vdash 0, \{i_1 \leftarrow v_1, \ldots, i_n \leftarrow v_m\}, []\xrightarrow{!}end, \sigma, []$.

On peut remarquer qu'il est assez facile de généraliser les machines à compteurs en utilisant n'importe quel type algébrique (par exemple les listes) au lieu des naturels. On obtient alors un langage qui est proche des langages "bytecode" utilisés lors de la compilation de langages fonctionnels.

Exemple 6.
Le programme suivant calcule la somme de deux entiers.

Entrées : x, y.

Sortie : y.

$$\begin{array}{rcl} 0 & : & \texttt{jz}\ x\ end \\ 1 & : & \texttt{dec}\ x \\ 2 & : & \texttt{inc}\ y \\ 3 & : & \texttt{jmp}\ 0 \\ end & & \end{array}$$

Définition 37 (Trace d'état).
Soit $(\texttt{lbl}, \sigma, \texttt{stk})$ un état de programme. Sa *trace d'état* $\theta(\texttt{lbl}, \sigma, \texttt{stk})$ est l'instruction à exécuter. Elle est formellement définie comme suit :

- Si $\texttt{instr}(\texttt{lbl}) \neq \texttt{jz}\ x\ \texttt{lbl}'$ alors $\theta(\texttt{lbl}, \sigma, \texttt{stk}) =_{\text{dfn}} \texttt{instr}(\texttt{lbl})$.
- Si $\texttt{instr}(\texttt{lbl}) = \texttt{jz}\ x\ \texttt{lbl}'$ et $\sigma(x) = 0$ alors $\theta(\texttt{lbl}, \sigma, \texttt{stk}) =_{\text{dfn}} x = 0$.
- Si $\texttt{instr}(\texttt{lbl}) = \texttt{jz}\ x\ \texttt{lbl}'$ et $\sigma(x) \neq 0$ alors $\theta(\texttt{lbl}, \sigma, \texttt{stk}) =_{\text{dfn}} x \neq 0$.

Définition 38 (Trace de programme).
Une *trace de programme* depuis un état s_0 est une suite finie ou infinie de traces d'état $\omega = \omega_0\omega_1 \ldots$ telle que $s_0 \rightarrow s_1 \rightarrow \ldots$ et pour tout i $\theta(s_i) = \omega_i$.

Si $\omega = \omega_0 \ldots \omega_n$ est une trace finie, alors on note $s_0\xrightarrow{\omega}s_{n+1}$.

1.3.3 Complexité

Définition 39 (Temps de calcul).
Soient pgm un programme et $v_1, \ldots, v_m$ des entiers naturels. Le *temps de calcul* de pgm sur $v_1, \ldots, v_m$ est égal à la longueur de la trace nécessaire pour normaliser $(0, \{i_1 \leftarrow v_1, \ldots, i_n \leftarrow v_n\}, [])$.

Définition 40 (Espace de calcul).
Soient pgm un programme et $v_1, \ldots, v_m$ des entiers. L'*espace de calcul* de pgm sur $v_1, \ldots, v_m$ est égal à la taille maximale d'une substitution obtenue dans un des états de la trace nécessaire pour normaliser le terme.

Les notions de complexité ne sont pas définies pour les machines à compteurs à cause de leur faible pouvoir d'expression.

En effet, elles comptent en unaire, ce qui introduit une différence fondamentale avec les autres modèles de calculs qui peuvent compter en binaire (ou plus) : la taille d'un entier unaire est égal à sa valeur, sinon elle est égale au logarithme de sa valeur. Ainsi, lors de la comparaison des machines à compteurs avec les autres modèles, il y aura toujours un facteur logarithmique (ou exponentiel, selon le sens) qui sera introduit. Ce phénomène est bien connu, par exemple quand on considère les machines de Turing dont l'alphabet n'a que deux symboles (et qui sont similaires aux machines à compteurs). De même, au chapitre 16 de son livre *Computability and Complexity from a Programming Perspective*, N. D. Jones [Jon97] remarque que les machines à compteurs n'ont pas assez de pouvoir d'expression pour donner une notion suffisamment intéressante de leur complexité.

Cependant, il faut tempérer cette remarque négative. En effet, il est assez facile de rajouter au langage une instruction $x := \texttt{half}(y)$ qui mettrait dans x le quotient de la division de y par 2 en laissant le reste dans y. À l'aide de cette instruction (et éventuellement d'une instruction de multiplication par 2), on peut considérer que les entiers sont stockés en binaire et on peut ainsi accéder à chaque bit pour les calculs, ce qui lève le problème de la représentation.

De même, il est facile d'étendre très légèrement le langage pour travailler sur des listes et non sur des entiers (`inc` et `dec` étant alors remplacés par **Cons** et `tail`). On obtient alors la représentation classique des listes et il n'y a plus de désavantage par rapport aux autres modèles de calculs.

1.4 Équivalence des modèles ?

1.4.1 Équivalence calculatoire

Thèse de Church.
Les trois modèles de calculs présentés précédemment sont équivalents. c'est-à-dire que toute fonction calculable par l'un des trois l'est aussi par les deux autres.

Tous les modèles de calculs permettent de calculer les mêmes fonctions.

La preuve complète et précise est relativement longue et complexe. Il s'agit d'un résultat classique de calculabilité. On donne ici quelques notions utiles pour la preuve. On peut lire pour voir les preuves complètes d'équivalence l'article de J. C. Shepherdson et H. E. Sturgis [SS63] pour l'équivalence des machines à compteurs et des machines de Turing. L'étude de N. Dershowitz [Der87] sur la terminaison des systèmes de réécriture montre la simulation d'une machine de Turing par un système de réécriture avec seulement 2 règles. Les chapitres 8 et 9 de [Jon97] donnent aussi des preuves complètes de ces équivalences.

Idées pour la preuve.
Le principal problème de ces simulations est de trouver un moyen d'encoder un modèle dans un autre.

Pour simuler une machine de Turing par un système de réécriture, on représente chacun des 4 mots d'une description instantanée par une liste de lettres. Ensuite, on dispose d'une fonction pour chaque état. La reconnaissance de motif se fait (en fonction de l'état) sur la lettre marquée dans chaque ruban (en se rappelant qu'une liste vide signifie en fait une infinité de $\sqcup$, ce qui génère beaucoup de cas particuliers à traiter proprement). Et on appelle la fonction correspondant au nouvel état, en mettant à jour les deux listes représentant chaque ruban (écriture d'une nouvelle lettre, déplacement de la tête de lecture).

Chaque pas de calcul de la machine de Turing est alors représenté par une réduction du système de réécriture. Pour l'état final, il faut éventuellement ajouter quelques étapes pour reconstruire le résultat à partir des deux listes qui le composent. Ceci demande un nombre linéaire de réductions. La complexité en temps du système de réécriture est donc un $O(n)$ de la complexité en temps de la machine de Turing. De même, les complexités en espace sont proportionnelles (si celle de la machine de Turing est assez grande, c'est-à-dire au moins linéaire, vu qu'elle n'est mesurée que sur le deuxième ruban).

Pour simuler un système de réécriture par une machine de Turing, on utilise un alphabet qui contient les symboles de fonctions, de constructeurs, des variables, des parenthèses, des virgules. On écrit sur le ruban de travail le terme à réduire et on le parcourt pour trouver le rédex et le réduire.

Pour simuler une machine à compteurs par une machine de Turing, on écrit sur le deuxième ruban les valeurs de chacun des compteurs, dans une base donnée, séparées par un symbole spécial. Les labels (et les instructions) sont codés dans les états de l'automate. Ensuite, il faut simuler chaque exécution d'une instruction, ce qui peut demander plusieurs pas de calculs pour trouver le compteur à modifier (ou à tester) et éventuellement (en cas d'incrément), déplacer tous les compteurs placés plus loin sur le ruban pour libérer la place nécessaire.

Pour simuler une machine de Turing par une machine à compteurs, on code chaque symbole de l'alphabet en binaire, puis chaque demi-ruban comme le nombre obtenu en mettant côte à côte les codes de tous les symboles le composant (de la même manière qu'on code des chaînes de caractères à l'aide du code ASCII). Il faut faire attention à garder les symboles qui sont modifiés pendant le calcul aux positions de poids faibles (de manière à n'avoir qu'une petite partie de chaque nombre à modifier à chaque fois). Ensuite, on dispose de 4 compteurs pour les rubans et un pour l'état et on peut simuler sans trop de difficultés la machine de Turing. $\square$

Définition 41 (Algorithme).
Un *algorithme* est soit une machine de Turing, soit un système de réécriture, soit une machine à compteurs. C'est-à-dire un moyen d'effectuer un calcul, sans préciser le modèle.

1.4.2 Extensionalité, intentionnalité

Puisque les trois modèles de calculs présentés ici permettent de calculer les mêmes fonctions, le mathématicien est content : ils sont équivalents. Mais le sont-ils vraiment ? Comme on l'a vu, la simulation d'un modèle par un autre engendre une perte de temps et d'espace. Cette perte est-elle nécessaire, n'y aurait-il pas moyen de calculer la même fonction dans chacun des modèles avec la même complexité ? Dans certains cas, on sait qu'il n'y a pas moyen de faire mieux. Par exemple, la simulation des machines RAM par des machines de Turing se fait obligatoirement avec une perte de temps.

Complexité explicite, complexité implicite

Prenons le problème un peu différemment en se concentrant sur une seule fonction avec un exemple assez classique : le tri. Il existe plusieurs algorithmes de tri. Tous n'ont pas la même complexité. On peut notamment citer le tri par insertion, qui trie une liste de longueur n en temps $O(n^2)$ et le tri par fusion qui trie cette même liste en temps $O(n\log(n))$. Donc si je cherche à calculer la fonction de tri, je vais avoir le choix entre plusieurs algorithmes, et je vais aboutir à des complexités différentes.

La première remarque que cette constatation appelle, c'est que la complexité qui a été définie est celle d'un *algorithme*, absolument pas d'une fonction. Dans un modèle de calculs donné, il peut exister plusieurs algorithmes qui calculent la *même* fonction.

Définition 42 (Complexité d'une fonction).
Soit f une fonction. La *complexité en temps* de f est la plus petite complexité en temps d'un algorithme qui calcule f.
La *complexité en espace* de f est la plus petite complexité en espace d'un algorithme qui calcule f.

La remarque concernant les complexités des machines de Turing ou des systèmes de réécriture s'applique aussi ici pour la comparaison des fonctions.

On peut maintenant calculer la complexité d'une fonction. Il est possible de montrer que la complexité de la fonction de tri est en $O(n\log(n))$ (voir par exemple le chapitre 9 de l'*Introduction à l'algorithmique* de T. Cormen, C. Leiserson et R. Rivest [CLR94]). Pour la plupart des autres fonctions, la complexité en temps n'est pas connue à l'heure actuelle. Montrer que la complexité d'un algorithme est optimale (c'est-à-dire calculer la complexité d'une fonction) est en règle générale très difficile à faire.

Il devient alors intéressant de renverser le point de vue. Au lieu de s'intéresser à la complexité des algorithmes qui calculent une fonction, on peut s'intéresser à la complexité de la fonction calculée par un algorithme donné. Ainsi, devant un tri par insertion, on peut certes dire que sa complexité est en $O(n^2)$. Mais on peut aussi dire que la complexité de la fonction qu'il calcule est en $O(n\log(n))$.

Définition 43 (Complexité implicite).
Soit A un algorithme calculant une fonction f. La *complexité explicite* de A est sa complexité telle que définie précédemment. La *complexité implicite* de A est la complexité de f.

Autrement dit, la complexité implicite d'un algorithme est la complexité de la fonction calculée par cet algorithme. L'idée qui se cache derrière cette définition est relativement simple. Est-ce que le programmeur qui a écrit un tri par insertion voulait vraiment écrire un programme "peu efficace" ? Ou est-ce, plus simplement, qu'il voulait *implicitement* écrire le "bon" algorithme mais en a été incapable (pour différentes raisons) ?

La réponse n'est pas évidente. En effet, on constate empiriquement que les "mauvais" algorithmes sont souvent plus simples à écrire, à relire, à comprendre ou à corriger que les "bons". En conséquence, dans un gros projet avec beaucoup de programmeurs, il peut être intéressant d'écrire un "mauvais" algorithme qui fera gagner tellement de temps en phases de développement, test, correction, ... que la perte de complexité du programme final sera compensée.

Et le but ultime de la transformation de programme apparaît ici : il s'agit, étant donné un algorithme, de trouver un autre algorithme, calculant la même fonction, mais avec une complexité optimale. On pourrait alors développer des projets en écrivant de manière compréhensible pour les humains, puis transformer le code inefficace ainsi obtenu en code efficace pour les ordinateurs mais illisible pour les humains.

Intentionnalité, extensionnalité

Mais revenons aux modèles de calculs. On a montré leur équivalence mathématique, c'est-à-dire qu'ils permettent de calculer les mêmes fonctions. Qu'en est-il du point de vue du programmeur ? Ces modèles permettent-il d'écrire les mêmes algorithmes ou, du moins, des algorithmes similaires ?

Si l'un des modèles ne permet d'écrire que le tri par insertion et non le tri fusion, est-il vraiment équivalent à un modèle qui autorise le tri fusion ? Bien sûr, les deux modèles permettent de calculer la même fonction (ici le tri). Mais l'un des deux est intrinsèquement moins efficace que l'autre.

La notion qui se dégage ici est extrêmement importante : pour le programmeur, la complétude d'un modèle en terme de fonctions calculées n'a aucune utilité. Il veut la complétude en terme d'algorithmes programmables et, plus particulièrement, il veut pouvoir écrire les "bons" algorithmes.

Cette notion, extrêmement importante, est celle de l'*intentionnalité* ou plus précisément de la *complétude intentionnelle* du modèle. Le problème n'est pas de savoir si le modèle peut calculer beaucoup de fonctions. Il est de savoir si il permet d'écrire les "bons" algorithmes pour calculer ces fonctions.

Dans le cadre des trois modèles présentés ici, l'intentionnalité ne sera pas discutée. Plus tard, ces modèles seront restreints par des contraintes syntaxiques supplémentaires (récursion primitive, ordres de terminaison, ...) afin de caractériser certaines classes de programmes (par exemple, ceux dont la complexité en temps est bornée par un polynôme). On s'apercevra alors qu'en voulant éliminer certaines fonctions du modèle, on a aussi éliminé certains algorithmes qu'on aurait bien aimé garder.

Notamment, L. Colson a montré [Col98] que le modèle de calcul de la récursion primitive permet de calculer la fonction min (qui renvoie le minimum de deux entiers) mais ne permet pas d'écrire l'algorithme qui calcule cette fonction en temps $O(min(x, y))$.

L'intentionnalité doit toujours rester présente à l'esprit quand on cherche à capturer des classes de fonctions dans un modèle de calcul. C'est mathématiquement très joli de réussir à capturer beaucoup de fonctions dans un petit modèle de calcul, mais c'est informatiquement complètement inutile. Ce qui est intéressant du point de vue informatique, c'est de capturer un grand nombre d'algorithmes couramment utilisés pour calculer ces fonctions.

Pour bien faire, il faudrait disposer d'un modèle de calculs qui soit, en quelque sorte, intentionnellement universel. C'est à dire qui puisse simuler tous les algorithmes de manière précise, donc simuler tous les autres modèles de calculs sans perte de temps. Si on admet la thèse de Y. Gurevich, il s'agirait des "Abstract States Machines" (ASM) [Gur99] qui, selon ses dires, peuvent simuler n'importe quel algorithme "à son niveau naturel d'abstraction".

Chapitre 2

Classes de complexité

Ce chapitre définit plusieurs classes de complexité, c'est-à-dire des classes de fonctions qui peuvent être calculées dans un temps ou un espace donné, qui est fonction des paramètres de la fonction. Ces classes sont montrées indépendantes du modèle de calculs choisi, en particulier pour ce qui concerne les modèles de calculs présentés au chapitre précédent. Définir les classes de complexité sur des fonctions est plus général que sur des problèmes de décision ou sur des ensembles. En effet, un problème de décision ou un ensemble peut toujours être ramené à sa fonction caractéristique.

J'ai choisi de présenter les classes par complexité croissante, en commençant par les plus petites (PTIME et LOGSPACE) et en terminant par les plus grandes (fonctions calculables ou semi-calculables). Ceci permet de présenter en premier les classes importantes pour le programmeur et pour l'analyse de programmes que sont les petites classes. En effet, savoir qu'un algorithme termine en temps élémentaire ou primitif récursif n'a que peu d'intérêt : l'algorithme est, en pratique, incalculable sur un ordinateur (il faut beaucoup trop de temps pour avoir le résultat).

2.1 La hiérarchie Espace/Temps

Non-déterminisme

Un algorithme (quel que soit le modèle de calculs) est non-déterministe si il peut à certains endroits faire un choix de manière aléatoire entre plusieurs possibilités.

Cela revient pour les machines de Turing à avoir une fonction de transition qui renvoie un ensemble d'états (un seul est choisi), pour les systèmes de réécriture, à être non confluent et pour les machines à compteurs, à avoir une instruction de saut aléatoire qui choisit un label destination parmi plusieurs.

Définition 44 (Algorithmes non-déterministes).
Une *machine de Turing non-déterministe* est un quintuplet (Σ, Q, q_0, q_f, F) où Σ, Q, q_0, q_f sont similaire aux machines déterministes et F est une fonction $Q \times \Sigma \times \Sigma \to Q^n \times \Delta \times \Delta \times \Sigma$ ($\Delta = \{\leftarrow, \downarrow, \rightarrow\}$). À chaque pas de calcul, le nouvel état de la machine est l'un de ceux renvoyés par la fonction de transition (choisi de manière aléatoire, équiprobable entre les différents choix possibles).

Un *système de réécriture non-déterministe* est un système de réécriture non-confluent.

Une *machine à compteurs non-déterministe* est une machine à compteur qui dispose d'une instruction de saut non-déterministe `jmp` $\texttt{lbl}_1$ *ou* $\texttt{lbl}_2$ qui choisit l'une des deux destinations de manière aléatoire.

La complexité en temps ou en espace d'un algorithme non-déterministe est égale à la complexité maximale pour chacune des exécutions possibles de l'algorithme.

La fonction calculée par un algorithme non-déterministe est celle dont la valeur en chaque point est égale au *maximum* des résultats de l'algorithme non-déterministe en ce point.

Le choix d'une valeur parmi les différents résultats possibles est nécessaire car je travaille sur des fonctions et non des problèmes de décision ou des ensembles caractéristiques. Ainsi, les différents résultats peuvent prendre n'importe quelle valeur et pas seulement 0 ou 1. Diverses solutions sont bien sûr possibles pour choisir un résultat et ont été étudiées notamment par D. Spreen [Spr88] ou N. Danner et C. Pollett [DP02]. La définition choisie ici s'inspire de celle de Y. Gurevich et E. Grädel [GG95]. Elle est en fait comparable à la théorie classique avec des problèmes de décision où seuls deux résultats (**True** et **False**) sont possibles et ordonnés **True** > **False**.

L'ordre de comparaison des résultats pour choisir le maximum est bien sûr à définir et le même ordre doit être utilisé pour chaque algorithme non-déterministe. En pratique la plupart des ordres "raisonnables" donnent les mêmes classes de complexité. On peut considérer pour fixer les idées qu'on prend un ordre lexicographique.

Exemple 7.

1. Supposons que l'on dispose d'une machine de Turing capable de tester si l'entier écrit (en binaire) sur le deuxième ruban est un diviseur non trivial de l'entier écrit sur le premier ruban. On peut alors construire facilement une machine non déterministe qui teste si un entier écrit sur le premier ruban est premier. En effet, il suffit d'écrire un nombre quelconque sur le deuxième ruban puis de tester si il s'agit d'un diviseur du nombre écrit sur le premier ruban. Si le nombre est premier, alors tous les calculs possibles renverront le résultat **faux**. Sinon, il y aura au moins un calcul qui trouvera un diviseur et renverra le résultat **True**.

   La partie "écrire un nombre quelconque" est réalisée par la machine suivante. Les états q_0, q_1 et q_2 écrivent sur le deuxième ruban un entier pas plus grand que celui écrit sur le premier. Pour cela, q_0 passe de manière non déterministe dans q_1 ou q_2 qui écrivent respectivement un **0** ou un **1** sur le deuxième ruban. Ensuite, la machine passe dans l'état q_3 qui repositionne les têtes au début de chaque mot. Puis l'état q_f correspond à l'état initial de la machine permettant de décider si un nombre est un diviseur d'un autre.
   - $\Sigma = \{\mathbf{0}, \mathbf{1}, \sqcup\}$
   - $Q = \{q_0, q_1, q_2, q_3, q_f\}$
   - F est donnée par le tableau (x, y désignent n'importe quelle lettre, **i** désigne **0** ou **1**) :

État	Lettre_1	Lettre_2	États	Mvt_1	Mvt_2	Écrire
q_0	$\sqcup$	y	q_3	$\leftarrow$	$\leftarrow$	y
q_0	$\mathbf{i}$	y	q_1, q_2	$\downarrow$	$\downarrow$	y
q_1	x	y	q_0	$\rightarrow$	$\rightarrow$	$\mathbf{0}$
q_2	x	y	q_0	$\rightarrow$	$\rightarrow$	$\mathbf{1}$
q_3	$\sqcup$	y	q_f	$\rightarrow$	$\rightarrow$	y
q_3	$\mathbf{i}$	y	q_3	$\leftarrow$	$\leftarrow$	y

2. Le système de réécriture suivant permet de décider la validité d'une expression booléenne. Il prend en entrée une formule booléenne composée de variables (représentées par des entiers) et des connecteurs **Or** et **Not** et, si il existe une assignation des variables qui permet de rendre vraie l'expression, renvoie une paire contenant **True** et une liste d'association représentant une telle assignation ; sinon, il renvoie une paire contenant **False** et une liste quelconque. Le système parcourt l'expression et à chaque fois qu'il rencontre une nouvelle variable, il lui assigne de manière non déterministe une valeur booléenne. Si l'expression est satisfiable, l'un des résultats du calcul sera **True**. Sinon, tous les résultats seront **False**.

   Les types sont $\mathtt{Nat}, \mathtt{Expr}, \mathtt{Pair}(\alpha, \beta), \mathtt{List}$. Les constructeurs sont $\mathcal{C} = \{\mathbf{0} : \mathtt{Nat}, \mathbf{S} : \mathtt{Nat} \rightarrow \mathtt{Nat}, \mathbf{True} : \mathtt{Expr}, \mathbf{False} : \mathtt{Expr}, \mathbf{Var} : \mathtt{Nat} \rightarrow \mathtt{Expr}, \mathbf{Not} : \mathtt{Expr} \rightarrow \mathtt{Expr}, \mathbf{Or} : \mathtt{Expr} \times \mathtt{Expr} \rightarrow \mathtt{Expr}, \bot : \mathtt{Expr}, \mathbf{Pair} : \alpha \times \beta \rightarrow \mathtt{Pair}(\alpha, \beta)\}$ et les fonctions sont définies par :

   $\mathtt{not} : \mathtt{Pair}(\mathtt{Expr}, \mathtt{List}) \rightarrow \mathtt{Pair}(\mathtt{Expr}, \mathtt{List})$

$$\mathtt{not}(\mathbf{Pair}(\mathbf{True}, \mathbf{l})) \rightarrow \mathbf{Pair}(\mathbf{False}, \mathbf{l})$$
$$\mathtt{not}(\mathbf{False}, \mathbf{l}) \rightarrow \mathbf{Pair}(\mathbf{True}, \mathbf{l})$$

   $\mathtt{or} : \mathtt{Expr} \times \mathtt{Pair}(\mathtt{Expr}, \mathtt{List}) \rightarrow \mathtt{Pair}(\mathtt{Expr}, \mathtt{List})$

$$\mathtt{or}(\mathbf{True}, \mathbf{Pair}(y, \mathbf{l})) \rightarrow \mathbf{Pair}(\mathbf{True}, \mathbf{l})$$
$$\mathtt{or}(x, y) \rightarrow y$$

$\texttt{\_=\_} : \texttt{Nat} \times \texttt{Nat} \to \texttt{Expr}$

$$
\begin{aligned}
\mathbf{0} = \mathbf{0} &\to \mathbf{True}\\
\mathbf{0} = \mathbf{S}(y) &\to \mathbf{False}\\
\mathbf{S}(x) = \mathbf{0} &\to \mathbf{False}\\
\mathbf{S}(x) = \mathbf{S}(y) &\to x = y
\end{aligned}
$$

$\texttt{ifte} : \texttt{Expr} \times \texttt{Expr} \times \texttt{Expr}$

$$
\begin{aligned}
\texttt{ifte}(\mathbf{True}, \varphi, \varphi') &\to \varphi\\
\texttt{ifte}(\mathbf{False}, \varphi, \varphi') &\to \varphi'
\end{aligned}
$$

$\texttt{in} : \texttt{Nat} \times \texttt{List} \to \texttt{Expr}$

$$
\begin{aligned}
\texttt{in}(x, \mathbf{Nil}) &\to \bot\\
\texttt{in}(x, \mathbf{Cons}(\mathbf{Pair}(y, \varphi), \mathbf{l})) &\to \texttt{ifte}(x = y, \varphi, \texttt{in}(x, \mathbf{l}))
\end{aligned}
$$

$\texttt{sat}_{\texttt{or}} : \texttt{Pair}(\texttt{Expr}, \texttt{List}) \times \texttt{Expr} \to \texttt{Pair}(\texttt{Expr}, \texttt{List})$

$$
\texttt{sat}_{\texttt{or}}(\mathbf{Pair}(x, \mathbf{l}), \varphi) \to \texttt{or}(x, \texttt{sat}(\varphi, \mathbf{l}))
$$

$\texttt{sat}_{\texttt{var}} : \texttt{Expr} \times \texttt{Nat} \times \texttt{List} \to \texttt{Pair}(\texttt{Expr}, \texttt{List})$

$$
\begin{aligned}
\texttt{sat}_{\texttt{var}}(\bot, x, \mathbf{l}) &\to \mathbf{Pair}(\mathbf{True}, \mathbf{Cons}(\mathbf{Pair}(x, \mathbf{True}), \mathbf{l}))\\
\texttt{sat}_{\texttt{var}}(\bot, x, \mathbf{l}) &\to \mathbf{Pair}(\mathbf{False}, \mathbf{Cons}(\mathbf{Pair}(x, \mathbf{False}), \mathbf{l}))\\
\texttt{sat}_{\texttt{var}}(\varphi, x, \mathbf{l}) &\to \mathbf{Pair}(\varphi, \mathbf{l})
\end{aligned}
$$

$\texttt{sat} : \texttt{Expr} \times \texttt{List} \to \texttt{Pair}(\texttt{Expr}, \texttt{List})$

$$
\begin{aligned}
\texttt{sat}(\mathbf{True}, \mathbf{l}) &\to \mathbf{Pair}(\mathbf{True}, \mathbf{l})\\
\texttt{sat}(\mathbf{False}, \mathbf{l}) &\to \mathbf{Pair}(\mathbf{False}, \mathbf{l})\\
\texttt{sat}(\mathbf{Not}(\varphi), \mathbf{l}) &\to \texttt{not}(\texttt{sat}(\varphi, \mathbf{l}))\\
\texttt{sat}(\mathbf{Or}(\varphi, \varphi'), \mathbf{l}) &\to \texttt{sat}_{\texttt{or}}(\texttt{sat}(\varphi, \mathbf{l}), \varphi')\\
\texttt{sat}(\mathbf{Var}(x), \mathbf{l}) &\to \texttt{sat}_{\texttt{var}}(\texttt{in}(x, \mathbf{l}), x, \mathbf{l})
\end{aligned}
$$

Le non déterminisme vient du fait que les règles de réécritures de $\texttt{sat}_{\texttt{var}}$ sont concurrentes. Elles peuvent être appliquées aux mêmes rédex avec des résultats (et des formes normales) différentes. En posant **True** $>$ **False** et $\varphi > \bot$ pour toute expression φ, on obtient bien un système non-déterministe qui renvoie une assignation des variables rendant l'expression vraie si c'est possible.

3. La machine à compteurs suivante calcule l'addition de deux nombres. En fait, chaque exécution donne un résultat inférieur ou égal à la somme souhaitée (ou une erreur). Comme on ne conserve que le plus grand résultat, et que la somme est effectivement obtenue pour un calcul, la machine non-déterministe permet de faire l'addition de deux nombres.
Entrées : x, y.
Sortie : y.

$$
\begin{array}{rcl}
0 & : & \texttt{jmp}\ 1\ \textit{ou end}\\
1 & : & \texttt{dec}\ x\\
2 & : & \texttt{inc}\ y\\
3 & : & \texttt{jmp}\ 0\ \textit{ou end}\\
\textit{end} & &
\end{array}
$$

La hiérarchie

Définition 45 (Temps polynômial).
Une fonction est dans PTIME si sa complexité en temps est bornée par un polynôme, c'est-à-dire en $O(n^a)$.

Une fonction est dans NPTIME si elle est calculable par un algorithme non-déterministe dont la complexité en temps est bornée par un polynôme.

Par abus de langage, les algorithmes pourront aussi être mis dans telle ou telle classe de complexité en fonction de leur complexité *explicite*.

On peut noter que la définition des classes de complexité dépend de la complexité des fonctions, donc de la complexité des algorithmes les calculant et donc, à priori, du ou des modèles de calculs choisis. On pourrait ainsi définir des classes de complexité restreintes à tel ou tel modèle de calcul pour le choix des algorithmes.

Définition 46 (Espace logarithmique).
Une fonction est dans LOGSPACE si sa complexité en espace (sur une machine de Turing) est en $O(log(n))$.

Une fonction est dans NLOGSPACE si elle est calculable par un algorithme non-déterministe dont la complexité est en $O(log(n))$.

Ces deux classes de complexité ne peuvent techniquement exister que pour le modèle des machines de Turing. En effet, la définition des autres modèles impose une complexité en espace minimale de n.

Définition 47 (Espace polynômial).
Une fonction est dans PSPACE si sa complexité en espace est bornée par un polynôme, c'est-à-dire en $O(n^a)$.

Une fonction est dans NPSPACE si elle est calculable par un algorithme non-déterministe dont la complexité en espace est en $O(n^a)$.

Définition 48 (Exponentielles).
Une fonction est dans E_kTIME si sa complexité en temps est bornée par une tour d'exponentielles de hauteur k, c'est-à-dire en $O(exp(exp(\ldots(exp(n))\ldots)))$ avec k exponentielles imbriquées.

Une fonction est dans NE_kTIME si elle est calculable par un algorithme non-déterministe de complexité bornée par une tour d'exponentielles de hauteur k.

Une fonction est dans E_kSPACE si sa complexité en espace est bornée par une tour d'exponentielles de hauteur k.

Les classes non-déterministes de complexité spatiale exponentielle ne sont pas définies car elles sont égales aux classes déterministes correspondantes.

La hiérarchie Espace/Temps.

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} \subseteq \text{NPSPACE}$$

Théorème de Savitch [Sav70].

$$\text{PSPACE} = \text{NPSPACE}$$

Suite de la hiérarchie Espace/Temps.

$$(N)\text{PSPACE} \subseteq \text{E}_1\text{TIME} \subseteq \text{NE}_1\text{TIME} \subseteq \text{E}_1\text{SPACE} \subseteq \ldots \subseteq \text{E}_k\text{TIME} \subseteq \text{NE}_k\text{TIME} \subseteq \text{E}_k\text{SPACE} \ldots$$

$$\text{LOGSPACE} \neq \text{PSPACE} \neq \text{E}_1\text{SPACE} \neq \ldots \neq \text{E}_k\text{SPACE} \ldots$$

Démonstration.
Tout algorithme déterministe étant aussi un algorithme non-déterministe (avec aucun endroit de choix possible), chaque classe de complexité déterministe est contenue dans la classe de complexité non-déterministe correspondante.

Un algorithme ne peut pas utiliser plus d'espace que son temps d'exécution (à une constante près). Ce qui donne l'inclusion des classes temporelles dans les classes spatiales correspondante.

Dans un espace E donné, il existe au maximum k^E configurations différentes possibles. Si un algorithme dont l'espace de calcul est borné par E tourne depuis plus de temps que k^E, il est forcément repassé au moins 2 fois par une configuration donnée et va donc partir en boucle infinie. Ce qui donne l'inclusion des classes spatiales dans les classes temporelles avec un décalage d'une exponentielle sur la complexité.

L'égalité PSPACE = NPSPACE entre les classes de complexité spatiales déterministes ou non est due à W. J. Savitch [Sav70].

Pour simuler un algorithme non-déterministe avec un algorithme déterministe, il faut, à chaque choix possible, garder la configuration en mémoire et explorer successivement toutes les possibilités. Ceci correspond à la technique de programmation dite du "backtracking". L'espace nécessaire pour stocker l'information utile au backtracking est polynomiale dans la taille des entrées. En effet, même si le nombre de calculs différents possibles est exponentiel, l'algorithme déterministe ne fait qu'un seul calcul à la fois. Il n'a donc besoin de stocker que l'information pour ce calcul, ainsi qu'un peu d'information pour pouvoir remonter en arrière. De même, l'algorithme non-déterministe a besoin de comparer un nombre exponentiel de résultats mais l'algorithme déterministe peut se contenter de garder en mémoire le plus grand des résultats déjà obtenus.

Le gain exponentiel d'espace permet de calculer strictement plus de fonctions (dernière ligne). Par exemple, la fonction qui prend une entrée de taille n et renvoie une sortie de taille 2^n est dans E_1SPACE mais pas dans PSPACE. Il est facile de construire sur ce modèle d'autres fonctions permettant de montrer les autres inégalités. □

À l'heure actuelle, on ne sait pas si les autres inclusions sont strictes ou non.

Savoir si l'inclusion LOGSPACE $\subseteq$ NLOGSPACE est stricte constitue le problème "$L = NL$?"

Savoir si l'inclusion PTIME $\subseteq$ NPTIME est stricte constitue le très célèbre problème "$P = NP$?" Ce problème joue un rôle très important dans l'informatique théorique. En effet on considère généralement qu'un problème soluble en temps polynômial est "faisable" en ce sens qu'il ne prendra pas trop de temps pour être résolu. Alors qu'un problème dans NPTIME n'est, à l'heure actuelle, soluble qu'en temps exponentiel, ce qui peut vite être très long (2^{100} opérations à $10GHz$ (10 milliards d'opérations par secondes) prennent plus d'un milliard d'années à s'effectuer).

Théorème 3.
À l'exception de LOGSPACE *et* NLOGSPACE*, les classes de complexité présentées sont les mêmes pour les machines de Turing et les systèmes de réécriture. C'est-à-dire si on n'autorise que les algorithmes d'un des deux modèle de calculs, on obtient les mêmes classes de complexités (en terme de fonctions)*

Démonstration.
Ces deux modèles se simulent mutuellement en temps polynômial. □

Remarque :
LOGSPACE est aussi caractérisé par les systèmes de réécriture récursifs terminaux qui ne comportent pas de constructeurs dans les membres droits des règles ([Jon00]).

2.2 Récursion primitive, récursion multiple

Les fonctions primitives récursives ont été introduites par T. Skolem en 1923 [Sko23]. La plupart du travail de développement a été fait par R. Péter et se retrouve dans son livre de 1966 [Pét66].

Définition 49 (Récursion primitive).
Un système de réécriture est *primitif récursif* si les règles sont de la forme :

$\mathtt{f}(p_1, \ldots, p_n) \to t$ où t ne contient que des symboles de fonctions définis précédemment.

ou $\mathtt{f}(\mathbf{0}, p_1, \ldots, p_n) \to \mathtt{g}(p_1, \ldots, p_n)$ où $\mathtt{g}$ est une fonction qui a été définie auparavant.

ou $\mathtt{f}(\mathbf{S}(x), p_1, \ldots, p_n) \to \mathtt{h}(x, \mathtt{f}(x, p_1, \ldots, p_n), p_1, \ldots, p_n)$ où $\mathtt{h}$ est un fonction définie préalablement.

Une fonction est *Récursive primitive* si elle est calculable par un système de réécriture primitif récursif.

PRIMREC est la classe de complexité qui contient toutes les fonctions primitives récursives.

Théorème 4.
Quel que soit k, $\mathrm{E}_k\mathrm{TIME} \subset \mathrm{PRIMREC}$.

Démonstration.
L'idée est de borner le temps de calcul d'une machine de Turing au moyen d'une horloge. On écrit ensuite un système de réécriture primitif récursif basé sur une fonction $\mathtt{f}$ telle que $\mathtt{f}(t, \omega, \omega', \nu, \nu')$ donne le résultat du calcul de la machine sur les rubans indiqués au bout de t pas de calcul.

Il est très facile de calculer $\mathtt{f}(t+1, \ldots)$ à partir de $\mathtt{f}(t, \ldots)$ (la fonction $\mathtt{h}$ utilisée pour définir le schéma récursif primitif simule le pas de calcul). Le temps de calcul total du système de réécriture est borné par la valeur initiale de l'horloge t.

Pour que l'ensemble du système soit primitif récursif, il suffit alors de pouvoir calculer la valeur initiale de l'horloge à partir de la taille des données de manière récursive primitive. Ceci ne pose aucun problème (on construit une série de fonctions calculant l'addition, la multiplication, l'exponentielle, puis les tours d'exponentielles de plus en plus hautes). □

Lors de l'introduction des fonctions primitives récursives, on a cru un moment que toutes les fonctions calculables étaient primitives récursives. W. Ackermann [Ack28] a prouvé le contraire en exhibant une fonction qui n'est pas primitive récursive et qui est calculée par le système de réécriture suivant :

$$\begin{array}{rcl} \mathtt{Ack}(\mathbf{0}, n) & \rightarrow & \mathbf{S}(n) \\ \mathtt{Ack}(\mathbf{S}(m), \mathbf{0}) & \rightarrow & \mathtt{Ack}(m, \mathbf{S}(\mathbf{0})) \\ \mathtt{Ack}(\mathbf{S}(m), \mathbf{S}(n)) & \rightarrow & \mathtt{Ack}(m, \mathtt{Ack}(\mathbf{S}(m), n)) \end{array}$$

La définition suivante est empruntée à H. E. Rose [Ros84].

Définition 50 (Multiples récursives).
Un système de réécriture est *k-récursif* si il est défini par des règles de la forme :
- $\phi(x, y_1, \ldots, y_k) \rightarrow 0$ si l'un des y_i est nul.
- $\phi(x, y_1 + 1, \ldots, y_k + 1) \rightarrow \mathtt{g}(x, y_1, \ldots, y_k, \phi_1, \ldots, \phi_k)$
  avec $\phi_i = \phi(x, y_1 + 1, \ldots, y_{i-1} + 1, y_i,$
  $\mathtt{f}_1^i(x, y_1, \ldots, y_k, \phi(x, y_1 + 1, \ldots, y_{k-1} + 1, y_k)), \ldots,$
  $\mathtt{f}_{k-i}^i(x, y_1, \ldots, y_k, \phi(x, y_1 + 1, \ldots, y_{k-1} + 1, y_k))$
  où $\mathtt{g}$ et les $\mathtt{f}_j^i$ ont été définis précédemment.

Une fonction est *k-récursive* si elle est calculable par un système de réécriture k-récursif. Une fonction est *multiple récursive* si elle est k-récursive (quelque soit k).

La classe $\mathrm{MULTREC}_k$ contient toutes les fonctions k-récursives. La classe MULTREC contient toutes les fonctions multiples récursives.

Théorème 5.
PRIMREC= $\mathrm{MULTREC}_1$.
$\mathrm{MULTREC}_k \in \mathrm{MULTREC}_{k+1}$.

2.3 Calculabilité

Définition 51.
La classe REC des fonctions *récursives* regroupe toute les fonctions calculables. c'est-à-dire toutes les fonctions pour lesquelles il existe un algorithme qui les calcule.

Théorème de la halte (A. Turing, 1936 [Tur36]).
Il existe des fonctions qui ne sont pas dans REC. *En particulier, la fonction* HALTE *qui prend en entrée un algorithme (quel que soit le modèle de calculs choisi) et dit si cet algorithme va terminer ou boucler infiniment, n'est pas calculable.*

On ne peut pas décider si un programme va s'arrêter ou boucler infiniment.

Démonstration.
Supposons qu'il existe un algorithme A qui calcule la fonction HALTE.

Construisons l'algorithme B de la manière suivante :

- B prend en entrée un algorithme a.
- B simule A sur cet algorithme. B sait donc si a termine ou pas.
- Si a termine, B part alors en boucle infinie. Sinon, B renvoie un résultat quelconque.

Quel est le comportement de B sur lui-même (en prenant $a = B$) ?

Si B termine, A le détecte, on est alors dans le premier cas et B doit boucler, donc ne pas terminer et on aboutit à une absurdité.

Si B ne termine pas, A le détecte, on est alors dans le deuxième cas et B termine, ce qui aboutit aussi à une absurdité.

Donc l'algorithme B ne peut pas exister. La seule hypothèse qui a été faite lors de sa construction est l'existence de A.

On en déduit donc que A ne peut pas exister, donc la fonction HALTE **n'est pas calculable**. □

Ce résultat est extrêmement décevant du point de vue de l'analyse de la complexité. En effet, la terminaison d'un algorithme est sans l'ombre d'un doute une propriété élémentaire qu'il serait extrêmement utile de connaître. On ne pourra jamais la connaître totalement. Mais il y a pire. Le **théorème de Rice** [Ric53] dit en substance que toute propriété intéressante (au sens de non-triviale) sur les algorithmes est aussi indécidable.

Définition 52 (Semi-calculable).
Une fonction f est *semi-calculable* si il existe une partition de son image en deux ensembles I_o et I_n et un algorithme A tel que :
- Pour tout entrée x telle que $f(x) \in I_o$, $A(x)$ termine et donne le bon résultat.
- Pour tout entrée y telle que $f(y) \in I_n$, $A(y)$ boucle infiniment.

Définition 53 (Récursivement énumerable).
La classe RE des fonctions *Récursivement Énumérables* regroupe toutes les fonctions semi-calculables.

Le nom "récursivement énumérable" vient du fait que si f est semi-calculable, il existe un algorithme A' qui prend en entrée les entiers naturels et qui renvoie les éléments de I_o. Ainsi, on peut *énumérer* au moyen d'une fonction *récursive* tous les éléments de I_o.

Proposition 6.
HALTE $\in$ *RE*.

Démonstration.
C'est une conséquence immédiate de la définition de la fonction HALTE. Il suffit de choisir pour I_o l'ensemble des résultats indiquant que l'algorithme donné en entrée termine et pour I_n l'ensemble des résultats indiquant qu'il boucle, puis de simuler l'algorithme. □

Théorème 7 (A. Turing, 1936 [Tur36]).
Il existe des fonctions qui ne sont pas semi-calculables.

Il existe des fonctions non semi-calculables.

Démonstration.
L'existence de fonctions non semi-calculables s'obtient assez facilement par un argument de cardinalité. En effet, l'ensemble des fonctions est non-dénombrable alors que l'ensemble des algorithmes est dénombrable. □

Une fonction non semi-calculable est celle qui dit si une fonction donnée est totale, c'est-à-dire termine pour *toutes* ses entrées.

Le théorème d'incomplétude de Gödel [Göd31] stipule qu'il est impossible d'établir une axiomatique qui permettrait de prouver *tous* les énoncés vrais dans l'arithmétique de Peano. La fonction qui prend comme entrée un énoncé de l'arithmétique et dit si il fait (ou non) partie de l'axiomatique n'est pas semi-calculable. La construction exacte est longue et délicate. Elle est décrite de manière assez précise dans le chapitre 6 du livre *Computational Complexity* de C. Papadimitriou [Pap94].

Théorème de Rice et intentionnalité

Revenons sur le théorème de Rice et sur ces conséquences pour l'analyse de programme. Comme il est impossible de décider une propriété intéressante pour l'ensemble des algorithmes, il va falloir trouver des classes d'algorithmes plus ou moins restreintes sur lesquelles des propriétés intéressantes pourront être décidées. Par exemple, on sait que les systèmes de réécriture primitifs récursifs terminent toujours.

Mais l'intentionnalité est à prendre en compte à ce moment. En effet, il ne sert à rien de prouver des propriétés intéressantes sur des ensembles d'algorithmes tellement restreints qu'ils ne nous apprennent rien. Par exemple, supposons que l'on veuille calculer la fonction min, qui prend 2 entiers en entrée et renvoie le minimum des deux en sortie. Le système de réécriture naturel pour y parvenir serait le suivant :

$$\begin{array}{rcl} \texttt{min}(\mathbf{0}, y) & \rightarrow & y \\ \texttt{min}(x, \mathbf{0}) & \rightarrow & x \\ \texttt{min}(\mathbf{S}(x), \mathbf{S}(y)) & \rightarrow & \mathbf{S}(\texttt{min}(x, y)) \end{array}$$

Ce système de réécriture a une complexité en temps en $O(min(x, y))$. La fonction min est primitive récursive (car elle est dans PTIME).

Or L. Colson a montré [Col98] que le système de réécriture ci-dessus n'était pas primitif récursif et qu'il n'existait **aucun** système de réécriture primitif récursif qui calcule min en temps $O(min(x, y))$. Ainsi, la classe des algorithmes primitifs récursifs ne contient pas le "bon" algorithme pour faire ce calcul. En conséquence, toutes les propriétés qu'on pourrait prouver sur les algorithmes primitifs récursifs perdent un peu de leur intérêt. En effet, à quoi bon faire des preuves sur des algorithmes qui ne seront jamais utilisés en pratique ?

Le travail de l'analyse de programmes devient donc doublement délicat. Il faut non seulement trouver des classes d'algorithmes suffisamment petites pour pouvoir dire des choses intéressantes dessus, mais il faut aussi que ces classes soient suffisamment grandes, suffisamment complètes *intentionnellement* pour regrouper des algorithmes qui sont utilisés en pratique.

Chapitre 3

Terminaison des systèmes de réécriture

Bien que, comme on l'a vu précédemment, la terminaison d'un système de réécriture soit indécidable dans le cas général, il existe des techniques permettant de prouver la terminaison de certaines classes de systèmes. Je vais ici présenter deux de ces techniques : les ordres de terminaisons et les interprétations polynomiales. Une autre technique extrêmement intéressante, le "Size-Change Principle" de C. S. Lee et al. [LJBA01] sera présentée plus tard.

Un aspect très intéressant de ces techniques est qu'elles "capturent" une des classes de complexité décrite au chapitre précédent. C'est-à-dire que pour chaque fonction appartenant à la classe de complexité, il existe un algorithme la calculant dont on peut prouver la terminaison avec la technique donnée.

La plupart des notions et notations utilisées ici sont reprises de l'étude "Terminaison of Rewriting" de N. Dershowitz [Der87].

3.1 Ordres de terminaison

Généralités

Définition 54 (Ordres de terminaison).
Soient $\mathcal{F}, \mathcal{C}$ deux ensembles de symboles et $\prec$ un ordre sur l'ensemble $\mathcal{T}(\mathcal{C}, \mathcal{F})$ des termes clos. $\prec$ est un *ordre de terminaison* si il possède les deux propriétés suivantes :

- $\prec$ est *bien-fondé*, c'est-à-dire qu'il n'existe pas de suite infinie décroissante de termes.
- $\prec$ est *monotone*, c'est-à-dire que si deux termes ne diffèrent que par un sous-terme, le plus petit est celui qui a le plus petit sous-terme (si il existe deux termes t et s tels que $t \prec s$ alors $\mathtt{f}(\ldots, t, \ldots) \prec \mathtt{f}(\ldots, s, \ldots)$).

Théorème 8 (D. S. Lankford, 1977, **[Lan77]).**
Soit main un système de réécriture. main termine si et seulement si il existe un ordre de terminaison $\prec$ tel que pour toute substitution σ et pour toute règle $l \to r$, $r\sigma \prec l\sigma$.
On dit alors que main termine par $\prec$.

Ce théorème se retrouve sous des formes similaires chez Z. Manna et S. Ness [MN70] ou S. Kamin et J.-J. Lévy [KL80].

Démonstration.
À chaque pas de réduction, un des sous-termes (le rédex) est remplacé par un sous-terme plus petit (car l'application d'une règle fait décroître l'ordre). Par monotonie de l'ordre, le terme décroît à chaque pas de réduction. Comme l'ordre est bien fondé, il ne peut pas y avoir de suite infinie de réductions, donc le système termine toujours. □

Définition 55 (Précédence).
Soient $\mathcal{F}, \mathcal{C}$ deux ensembles de symboles de fonctions et de constructeurs. Une *précédence* $\preceq_\Sigma$ est un ordre sur $\Sigma = \mathcal{F} \bigcup \mathcal{C}$.

Un ordre $\preceq_\mathcal{F}$ sur $\mathcal{F}$ est considéré comme une précédence en posant $\mathbf{c} \prec_\mathcal{F} \mathtt{f}$ pour tout $\mathbf{c} \in \mathcal{C}$ et $\mathtt{f} \in \mathcal{F}$ et $\mathbf{c} \approx_\mathcal{F} \mathbf{c}'$.

Définition 56 (Ordres sur les chemins).
Soient $\preceq_{\mathcal{F}}$ une précédence et $\bullet^*$ une manière de prolonger un ordre sur les termes en ordre sur les listes de termes (c'est-à-dire que si $\prec$ est un ordre sur les termes, $\prec^*$ est un ordre sur les listes de termes). On associe à $\bullet^*$ l'*ordre sur les chemins* $\blacktriangleleft$ tel que $t = \mathtt{f}(t_1, \ldots, t_n) \blacktriangleleft \mathtt{g}(s_1, \ldots, s_m) = s$ si l'une des conditions suivantes est vraie :

- Il existe j tel que $t \blacktriangleleft s_j$.
- $\mathtt{f}, \mathtt{g} \in \mathcal{C}, m = n$ et il existe une permutation π telle que pour tout j, $t_j \blacktriangleleft s_{\pi(j)}$ ou $t_j = s_{\pi(j)}$ et il existe i tel que $t_i \blacktriangleleft s_{\pi(i)}$.
- $\mathtt{f} \prec_{\mathcal{F}} \mathtt{g}$ et $t_i \blacktriangleleft s$ pour tout i.
- $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, $\mathtt{f}, \mathtt{g} \in \mathcal{F}$, $(t_1, \ldots, t_n) \blacktriangleleft^* (s_1, \ldots, s_m)$ et pour tout j, $t_j \blacktriangleleft s$.

Des deux ordres sur les chemins que je présente, LPO et MPO, je choisis de commencer par LPO car il est basé sur l'ordre lexicographique qui est plus facile à la fois à comprendre et à expliquer que l'ordre multiset sur lequel est basé MPO. Ceci permet, je l'espère, de se concentrer sur les problèmes spécifiques aux ordres sur les chemins sans être encombré par les problèmes spécifiques à l'ordre multiset.

3.1.1 Lexicographic Path Ordering

Définition 57 (Ordre lexicographique).
Soit $\prec$ un ordre sur un ensemble E. Son *prolongement lexicographique* $\prec^l$ est un ordre sur les listes d'éléments de E tel que $(a_1, \ldots, a_n) \prec^l (b_1, \ldots, b_m)$ si l'une des conditions suivantes est vraie :

- $a_1 \prec b_1$.
- $a_1 = b_1$ et $(a_2, \ldots, a_n) \prec^l (b_2, \ldots, b_m)$.

Lemme 9.
Soient $\prec$ un ordre bien-fondé et n un entier. Sur l'ensemble des listes de longueur inférieure ou égale à n, le prolongement lexicographique de $\prec$ est bien-fondé.

Si on autorise les listes de tailles arbitraires, on peut facilement construire une suite infinie décroissante comme par exemple (sur les entiers ordonnés de manière usuelle) : $(1) \succ (0,1) \succ (0,0,1) \succ \ldots \succ (0,0,\ldots,0,1) \succ \ldots$

Démonstration.
Par récurrence sur n. Si $n = 1$, $\prec^l = \prec$ est bien-fondé. Sinon, $\prec$ étant bien-fondé on a un nombre fini de listes où la décroissance porte sur le premier élément puis, par hypothèse de récurrence, un nombre fini où la décroissance porte sur les $n-1$ derniers éléments. $\square$

Définition 58 (LPO).
L'ordre lexicographique sur les chemins (*LPO*, *Lexicographic Path Ordering*) est l'ordre sur les chemins associé au prolongement lexicographique. c'est-à-dire qu'on a $t = \mathtt{f}(t_1, \ldots, t_n) \prec_{lpo} \mathtt{g}(s_1, \ldots, s_m) = s$ ($\mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$) si l'une des conditions suivantes est vraie :

- Il existe j tel que $t \preceq_{lpo} s_j$.
- $\mathtt{f}, \mathtt{g} \in \mathcal{C}, m = n$ et il existe une permutation π telle que pour tout j, $t_j \preceq_{lpo} s_{\pi(j)}$ et il existe i tel que $t_i \prec_{lpo} s_{\pi(i)}$.
- $\mathtt{f} \prec_{\mathcal{F}} \mathtt{g}$ et $t_i \prec_{lpo} s$ pour tout i.
- $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, $\mathtt{f}, \mathtt{g} \in \mathcal{F}$, $(t_1, \ldots, t_n) \prec^l_{lpo} (s_1, \ldots, s_m)$ et pour tout j, $t_j \prec_{lpo} s$.

Théorème 10 (S. Kamin & J.-J. Levy [KL80]).
LPO est un ordre de terminaison.

Exemple 8.
Le système de réécriture suivant, qui calcule la fonction d'Ackermann, termine par LPO.

$$
\begin{array}{rcl}
\mathtt{Ack}(\mathbf{0}, n) & \rightarrow & \mathbf{S}(n) \\
\mathtt{Ack}(\mathbf{S}(m), \mathbf{0}) & \rightarrow & \mathtt{Ack}(m, \mathbf{S}(\mathbf{0})) \\
\mathtt{Ack}(\mathbf{S}(m), \mathbf{S}(n)) & \rightarrow & \mathtt{Ack}(m, \mathtt{Ack}(\mathbf{S}(m), n))
\end{array}
$$

Théorème 11 (A. Weiermann [Wei95]).
Chaque système de réécriture qui termine par LPO calcule une fonction multiple récursive et chaque fonction multiple récursive est calculable par (au moins) un système de réécriture qui termine par LPO.

$$\boxed{\text{LPO} \equiv \text{MultRec.}}$$

3.1.2 Multiset Path Ordering

Définition 59 (Ordre multiset).
Soit $\prec$ un ordre sur un ensemble E. Son *prolongement multiset* $\prec^m$ est un ordre sur les listes d'éléments de E tel que $A = (a_1, \ldots, a_n) \preceq^m (b_1, \ldots, b_m) = B$ si l'une des conditions suivantes est vraie :
- $A = B$ ou A est vide.
- Il existe $a_j \equiv b_i$ et $A - \{a_j\} \preceq^m B - \{b_i\}$.
- Il existe $1 \leq i \leq m$ et $1 \leq j_1 < j_2 < \ldots < j_k \leq n$ tels que $b_i \succ a_{j_1}, \ldots, a_{j_k}$ et $A - \{a_{j_1}, \ldots, a_{j_k}\} \preceq^m B - \{b_i\}$.

Exemple 9.
Sur des multisets de naturels, on a les relations suivantes :
- $(3, 4) <^m (3, 3, 4, 0)$ car $(3) <^m (3, 3, 0)$ (deuxième règle) car $() <^m (3, 0)$ (deuxième règle) car le plus petit multiset est vide.
- $(3, 2, 2, 2, 1, 1, 1, 4, 0) <^m (3, 3, 4, 0)$ car $(3, 2, 2, 2, 1, 1, 1, 0) <^m (3, 3, 0)$ (deuxième règle) car $(3) <^m (3, 0)$ (troisième règle) car $() <^m (0)$ (multiset vide).
- $(3, 3, 3, 3, 2, 2) <^m (3, 3, 4, 0)$.

Une autre façon de voir l'ordre multiset est la suivante : on classe chacun des deux multisets selon l'ordre $\prec$ (en ordre décroissant) et on compare ensuite (par le prolongement lexicographique de l'ordre sur $\mathbb{N}$) les listes composées du *nombre d'occurrences* de chaque élément dans chaque multiset.

Exemple 10.
Les trois comparaisons précédentes deviennent donc :
- $(3, 4) <^m (3, 3, 4, 0)$ car $(1, 1, 0, 0, 0) <^l (1, 2, 0, 0, 1)$.
- $(3, 2, 2, 2, 1, 1, 1, 4, 0) <^m (3, 3, 4, 0)$ car $(1, 1, 3, 3, 1) <^l (1, 2, 0, 0, 1)$.
- $(3, 3, 3, 3, 2, 2) <^m (3, 3, 4, 0)$ car $(0, 4, 2, 0, 0) <^l (1, 2, 0, 0, 1)$.

Lemme 12.
Si $\prec$ est un ordre bien-fondé, son prolongement multiset $\prec^m$ est bien-fondé.

C'est cette propriété qui permet à Hercule de tuer l'hydre quel que soit la stratégie adoptée. En effet, l'hydre fait repousser ses têtes plus près du corps que celle qu'Hercule a coupée et l'ordre multiset sur la distance des têtes au corps décroît à chaque coup.

Définition 60 (MPO).
L'ordre multiset sur les chemins (*MPO, Multiset Path Ordering*) est l'ordre sur les chemins associé au prolongement multiset. c'est-à-dire qu'on a $t = \mathtt{f}(t_1, \ldots, t_n) \prec_{mpo} \mathtt{g}(s_1, \ldots, s_m) = s$ ($\mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$) si l'une des conditions suivantes est vraie :
- Il existe j tel que $t \preceq_{mpo} s_j$.
- $\mathtt{f}, \mathtt{g} \in \mathcal{C}, m = n$ et il existe une permutation π telle que pour tout j, $t_j \preceq_{mpo} s_{\pi(j)}$ et il existe i tel que $t_i \prec_{mpo} s_{\pi(i)}$.
- $\mathtt{f} \prec_{\mathcal{F}} \mathtt{g}$ et $t_i \prec_{mpo} s$ pour tout i.
- $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, $\mathtt{f}, \mathtt{g} \in \mathcal{F}$ et $(t_1, \ldots, t_n) \prec^m_{mpo} (s_1, \ldots, s_m)$.

Remarque :
Dans certains articles, MPO désigne le "Monadic Path Ordering" et l'ordre décrit ici est appelé RPO (Recursive Path ordering). Comme l'expliquent N. Dershowitz et J.-P. Jouannaud [DJ90], RPO désigne maintenant un ordre qui mélange les ordres lexicographiques et multiset (selon le symbole de fonction).

Théorème 13 (N. Dershowitz, 1982 [Der82]).
MPO est un ordre de terminaison.

Théorème 14 (A. Cichon, 1990, et D. Hofbauer, 1992 [Cic90, Hof92]).
Chaque fonction primitive récursive est calculable par un système de réécriture qui termine par MPO et chaque système de réécriture qui termine par MPO calcule une fonction primitive récursive.

$$\boxed{\text{MPO} \equiv \text{PrimRec}.}$$

Il existe aussi d'autres ordres de terminaison. On peut en particulier citer MPO' qui est une restriction de MPO que je présente au chapitre suivant, RPO (Recursive Path Ordering) qui est un mélange de LPO et de MPO (dans le cas où les symboles de tête ont la même précédence, les sous-termes sont comparés soit avec l'extension lexicographique, soit avec l'extension multiset, selon le statut de la fonction) ou KBO (Knuth-Bendix Ordering) introduit par D. E. Knuth et P. B. Bendix [KB70].

3.2 Interprétations polynomiales

Les interprétations polynomiales sont un cas un peu particulier d'ordre de terminaison. En effet, on commence par transformer les termes en entiers naturels et ensuite on s'assure de la terminaison du système grâce à la décroissance des entiers lors des réductions (l'ordre sur les naturels étant bien fondé on ne peut avoir de chaîne de réduction infinie).

Les interprétations polynomiales ont été introduites par D. S. Lankford [Lan79]. Bien que l'existence d'une interprétation polynomiale (sur les entiers) soit indécidable (c'est une conséquence du $10^{\text{ème}}$ problème de Hilbert), de nombreuses heuristiques permettent d'obtenir des bons résultats. Plusieurs systèmes de preuve de terminaison par interprétations polynomiales ont ainsi été bâtis, comme par exemple les systèmes `Reve` [CL87], `Polo` [Gie95] ou `Larch` [Lar].

De plus, D. Hofbauer et C. Lautemann [HL88] ont montré que la complexité des systèmes de réécritures terminant par interprétation polynomiale est au plus doublement exponentielle, ce qui permet de capturer des classes de complexité beaucoup plus petites (et donc beaucoup plus faisables en pratique) que ce qu'on obtenait avec les ordres de terminaison. G. Bonfante, A. Cichon, J.-Y. Marion et H. Touzet [BCMT98, BCMT00] ont affiné le résultat précédent en montrant que la forme de l'interprétation polynomiale joue un rôle très important dans la complexité du système.

Définition 61 (Interprétation polynômiale).
L'*interprétation polynomiale* d'un symbole $a \in \mathcal{F} \bigcup \mathcal{C}$ d'arité n est un polynôme à n variables $[\![a]\!]$ tel que :

1. $[\![a]\!](X_1, \dots, X_n) > \sum_{i=1}^{n} X_i$.
2. $[\![a]\!]$ est strictement croissant vis à vis de chacune des variables.
3. L'interprétation des constantes (constructeurs d'arité 0) est strictement positive.

Les interprétations sont étendues aux termes clos de manière canonique :

$$[\![a(t_1, \dots, t_n)]\!] = [\![a]\!]([\![t_1]\!], \dots, [\![t_n]\!])$$

Un système de réécriture *admet* une interprétation polynomiale si pour chaque règle $r \to l$ et pour chaque substitution close σ, $[\![r\sigma]\!] > [\![l\sigma]\!]$.

Proposition 15.
Soit main un système de réécriture qui admet une interprétation polynomiale. L'ordre induit sur les termes :

$$t \prec s \iff [\![t]\!] < [\![s]\!]$$

est un ordre de terminaison.

Démonstration.
L'ordre est bien fondé car $[\![t]\!]$ est toujours positif et l'ordre sur $\mathbb{N}$ est bien fondé. La monotonie découle de la croissance des interprétations. $\square$

Corollaire 16.
Un système de réécriture qui admet une interprétation polynomiale termine.

Définition 62 (Genre des polynômes et des systèmes).
Un polynôme $P(X_1, \ldots, X_n)$ est de

genre 0 si $P(X_1, \ldots, X_n) = \sum_{i=1}^{n} X_i + b$, b constante strictement positive ;

genre 1 si P est un polynôme de degré 1 dans chacun de ses variables ;

genre 2 si P est un polynôme quelconque.

Un système de réécriture confluent (resp. non-confluent) admettant une interprétation polynomiale est de *genre* $\Pi(i)$ (resp. $\Delta(i)$) si l'interprétation des constructeurs est de genre i.

Théorème 17 (G. Bonfante, A. Cichon, J.-Y. Marion, H. Touzet 1998 **[BCMT98]).**
Les systèmes de réécriture admettant une interprétation polynomiale caractérisent différentes classes de complexité selon leur genre.

Genre du système	*Classe caractérisée*
$\Pi(0)$	PTIME
$\Pi(1)$	E_1TIME
$\Pi(2)$	E_2TIME

Genre du système	*Classe caractérisée*
$\Delta(0)$	NPTIME
$\Delta(1)$	NE_1TIME
$\Delta(2)$	NE_2TIME

Chapitre 4

Une caractérisation de PTIME

PTIME est souvent considérée (par exemple par S. Cook [Coo91]) comme étant la classe des fonctions effectivement calculables (en temps raisonnable) par un ordinateur. Même si, comme le pensent plusieurs personnes, ceci n'est pas toujours exact (on peut trouver une discussion à ce sujet dans [Gur93]), PTIME a été étudiée de près et plusieurs caractérisations algorithmiques de cette classe ont vu le jour. La première d'entre elle est due à A. Cobham [Cob62] et utilise des fonctions "annexes" pour borner la complexité des fonctions calculées. L'étude de N. D. Jones sur les programmes fonctionnels ne pouvant pas écrire [Jon01] a montré que de tels programmes restreints à l'ordre 0 caractérisent PTIME alors que les ordres supérieurs caractérisent E_kTIME.

Une autre idée très intéressante est celle de la séparation des données ("data tiering") ou analyse prédicative. L'idée a été introduite par H. Simmons [Sim88], développée par S. Bellantoni et S. Cook [BC92, Bel94] et largement reprise par D. Leivant et J.-Y. Marion [Lei94, LM95, LM97, LM00b, LM00a, Lei99], H. Schwichtenberg [BNS00] ou M. Hofmann [Hof97, BH00].

4.1 Analyse prédicative

L'analyse prédicative d'un système de réécriture consiste à séparer les arguments des fonctions suivant deux niveaux ("tiers"), normal ou sûr ("normal"/"safe"). De manière très informelle, les arguments normaux peuvent être utilisés pour contrôler une récurrence mais pas les arguments sûrs. La valeur renvoyée par une fonction définie par récurrence doit toujours être utilisée en position sûre et ne peut donc pas servir à contrôler une nouvelle récurrence. Une autre façon de voir les choses est de considérer que les arguments ont de l'énergie, qu'il faut de l'énergie pour faire une récurrence (au moins un argument ayant encore de l'énergie doit décroître) et que l'énergie est consommée par cette récurrence.

En injectant l'analyse prédicative dans les ordres de terminaison, on obtient les ordres de terminaisons allégés qui ont été étudiés par J.-Y. Marion et A. Cichon [Mar00a, CM00].

4.1.1 Une variante de MPO

Définition 63 (Ordre de permutations).
Soit $\prec$ un ordre sur un ensemble E. Son *prolongement par permutations* $\prec^{m'}$ est un ordre sur les listes d'éléments de E de même longueur tel que $A = (a_1, \ldots, a_n) \preceq^{m'} (b_1, \ldots, b_n) = B$ si il existe une permutation π telle que :

- Pour tout i, $a_i \preceq b_{\pi(i)}$.
- Il existe j tel que $a_j \prec b_{\pi(j)}$

Lemme 18.
Le prolongement par permutations est une restriction du prolongement multiset. c'est-à-dire que $A \prec^{m'} B$ implique $A \prec^{m} B$.

Corollaire 19.
Si $\prec$ est un ordre bien-fondé, son prolongement par permutations $\prec^{m'}$ est bien-fondé.

Définition 64 (MPO').
L'ordre *MPO'* est l'ordre sur les chemins associé au prolongement par permutations. c'est-à-dire qu'on a $t = \mathtt{f}(t_1, \dots, t_n) \prec_{mpo'} \mathtt{g}(s_1, \dots, s_n) = s$ si l'une des conditions suivantes est vraie :
- Il existe j tel que $t \preceq_{mpo'} s_j$.
- $\mathtt{f} \prec_{\mathcal{F}} \mathtt{g}$ et $t_i \prec_{mpo'} s$ pour tout i.
- $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ et $(t_1, \dots, t_n) \prec^{m'}_{mpo'} (s_1, \dots, s_n)$.

Lemme 20 (J.-Y. Marion et J.-Y. Moyen, 2000 **[MM00]).**
MPO' est une restriction de MPO.

Corollaire 21.
MPO' est un ordre de terminaison.

Théorème 22 (J.-Y. Marion et J.-Y. Moyen, 2000 **[MM00]).**
Chaque fonction primitive récursive est calculable par un système de réécriture qui termine par MPO' et chaque système de réécriture qui termine par MPO' calcule une fonction primitive récursive.

Démonstration.
MPO' étant une restriction de MPO, tout système de réécriture qui termine par MPO' termine aussi par MPO et calcule donc une fonction primitive récursive. Réciproquement, il est facile de voir que toutes les règles autorisées par le schéma primitif récursif terminent par MPO'. □

4.1.2 Light Multiset Path Ordering

LMPO est la variante avec analyse prédicative de MPO'.

Définition 65 (Valence).
La *valence* d'un symbole de fonction $\mathtt{f}$ d'arité n est une fonction $\nu(\mathtt{f}) : \{1, \cdots, n\} \mapsto \{0, 1\}$.

La valence est en quelque sorte une généralisation des statuts de S. Kamin et J.-J. Levy [KL80] et indique comment doivent être comparés les arguments des divers symboles de fonctions.

Les valences des arguments sont similaires aux niveaux de S. Bellantoni et S. Cook. Un argument de valence 1 est *normal*, il peut être utilisé pour contrôler une récurrence. Un argument de valence 0 est *sûr*, il ne peut pas être utilisé pour contrôler une récurrence, c'est-à-dire que lors d'une récurrence il doit toujours y avoir au moins un argument de valence 1 qui décroît. Le résultat des fonctions ne peut être utilisé qu'en position sûre (de valence 0).

C'est en ce sens que la valence peut être assimilée à de l'énergie contenue dans les arguments. Pour faire une récurrence, il faut consommer de l'énergie (faire décroître un argument de valence 1) et une fois celle-ci finie, toute l'énergie a été consommée et le résultat est de valence 0.

Définition 66.
Soit ν une fonction de valence sur $\mathcal{F}$. Une permutation π *respecte les valences* de $\mathtt{f}$ et $\mathtt{g}$ si les deux symboles de fonctions ont la même arité n et que pour tout i, $\nu(\mathtt{g}, i) = \nu(\mathtt{f}, \pi(i))$.

Le prolongement par permutations est modifié de manière à prendre en compte les valences. Il faut maintenant que l'élément qui décroisse soit un élément de valence 1.

Définition 67 (Ordre par permutations et valence).
Soient $\prec_0$ et $\prec_1$ deux ordres sur les termes. Ces ordres sont prolongés en un ordre $\prec^\nu$sur les listes de termes qui respecte une fonction de valence ν définie sur $\mathcal{F}$.

$(s_1, \dots, s_n) \prec^{\nu}_{\mathtt{g},\mathtt{f}} (t_1, \dots, t_n)$ si et seulement si il existe une permutation π qui respecte la valence de $\mathtt{f}$ et $\mathtt{g}$ et qui vérifie :
- Pour tout i, $s_i \preceq_{\nu(\mathtt{g},i)} t_{\pi(i)}$.
- Il existe j tel que $\nu(\mathtt{g}, j) = 1$ et $s_j \prec_1 t_{\pi(j)}$.

L'ordre ainsi défini est une restriction du prolongement par permutations.

Définition 68 (LMPO).
Soit $\prec_{\mathcal{F}}$ une précédence sur $\mathcal{F}$. L'*ordre multiset sur les chemins allégé* (*LMPO*, Light Multiset Path Ordering) est une paire d'ordres $(\prec_k)_{k=0,1}$ définis comme suit :

1. $s \prec_k t$ si il existe un sous-terme $u \lhd t$ tel que $s \preceq_k u$.
2. $\mathtt{g}(s_1, \ldots, s_n) \prec_k \mathtt{f}(t_1, \ldots, t_m)$ si $(\mathtt{g} \prec_{\mathcal{F}} \mathtt{f})$ et si $s_i \prec_{\max(k,\nu(\mathtt{g},i))} \mathtt{f}(t_1, \ldots, t_m)$ pour chaque $i \leq n$.
3. $\mathtt{g}(s_1, \ldots, s_n) \prec_0 \mathtt{f}(t_1, \ldots, t_n)$ si $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ et $\{s_1, \ldots, s_n\} \prec^{\nu}_{\mathtt{g},\mathtt{f}} \{t_1, \ldots, t_n\}$

$(\mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C})$.

La troisième règle stipule que lors d'une récurrence (lorsqu'on compare deux termes avec le même symbole de tête), il faut absolument qu'un argument de valence 1 décroisse. De plus, comme cette comparaison n'est possible qu'avec l'ordre $\prec_0$, elle ne peut avoir lieu qu'en position de valence 0. Ainsi, le résultat de la récurrence est forcément en position de valence 0.

Le couple d'ordres ainsi défini forme un ordre de terminaison.

Définition 69.
Un système de réécriture *termine par LMPO* si il existe une précédence et une fonction de valence telles que pour chaque règle $l \to r$ et pour chaque substitution close σ, $r\sigma \prec_0 l\sigma$.

Théorème 23 (J.-Y. Marion, 2000 **[Mar00a]).**
LMPO capture PTIME. *C'est-à-dire que chaque système de réécriture qui termine par LMPO calcule une fonction de* PTIME *et que chaque fonction de* PTIME *est calculable par un système de réécriture qui termine par LMPO.*

$$\boxed{\text{LMPO} \equiv \text{PTIME}.}$$

Il convient ici de faire attention au sens des implications. En effet, ce n'est pas parce qu'un système ne termine pas par LMPO qu'il ne calcule pas une fonction de PTIME. Ainsi, on ne peut pas vraiment espérer utiliser LMPO pour prouver PTIME $\neq$ NPTIME. En effet, si un système de réécriture calculant une fonction de NPTIME ne termine pas par LMPO, cela ne veut pas dire que la fonction calculée n'est pas dans PTIME.

Exemple 11 (Complexité implicite et intentionnalité).
Le système de réécriture suivant permet de calculer la plus longue sous-suite commune (plssc) à deux suites de lettres. On a deux suites X et Y de lettres et on cherche la longueur de la plus longue suite qui soit extraite à la fois de X et Y. Par exemple, si $X = abaabb$ et $Y = aababa$, la plus longue sous-suite commune est $abab$, de longueur 4.

Par soucis de clarté, les fonctions ont été typées et les valences indiquées dans les types. De plus, tous les arguments normaux (de valence 1) sont groupés en tête du terme et séparés des arguments sûrs (de valence 0) par un point-virgule. Les mots binaires sont générés à l'aide des constructeurs $\{\epsilon : \mathtt{Word}, \mathbf{a} : \mathtt{Word} \to \mathtt{Word}, \mathbf{b} : \mathtt{Word} \to \mathtt{Word}\}$.

$\mathtt{max} : \mathtt{Word}_1, \mathtt{Nat}_0, \mathtt{Nat}_0 \to \mathtt{Nat}$.

$$\begin{aligned}
\mathtt{max}(x; n, \mathbf{0}) &\to n \\
\mathtt{max}(x; \mathbf{0}, m) &\to m \\
\mathtt{max}(\mathbf{i}(x); \mathbf{S}(n), \mathbf{S}(m)) &\to \mathbf{S}(\mathtt{max}(x; n, m)) && \mathbf{i} \in \{\mathbf{a}, \mathbf{b}\}
\end{aligned}$$

$\mathtt{main} = \mathtt{lcs} : \mathtt{Word}_1, \mathtt{Word}_1 \to \mathtt{Word}$.

$$\begin{aligned}
\mathtt{lcs}(\epsilon, \epsilon;) &\to \mathbf{0} \\
\mathtt{lcs}(\epsilon, y;) &\to \mathbf{0} \\
\mathtt{lcs}(x, \epsilon;) &\to \mathbf{0} \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{i}(y);) &\to \mathbf{S}(\mathtt{lcs}(x, y;)) \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{j}(y);) &\to \mathtt{max}(\mathbf{i}(x); \mathtt{lcs}(x, \mathbf{j}(y);), \mathtt{lcs}(\mathbf{i}(x), y;)) && \mathbf{i} \neq \mathbf{j}
\end{aligned}$$

Le système obtenu termine par LMPO en mettant $\mathtt{max} \prec_{\mathcal{F}} \mathtt{lcs}$. La fonction calculée appartient donc à PTIME.

Cet exemple appelle deux remarques extrêmement importantes.

Tout d'abord sur la construction de max. Le premier argument n'a aucune valeur calculatoire. Sa seule utilité est de faire décroître un argument de valence 1 dans la dernière règle de max. En effet, sans sa présence, il faudrait que n ou m soit de valence 1. Or ceci est impossible à cause de la troisième règle de lcs. m et n sont le résultat d'une fonction (lcs) et sont donc forcément de valence 0. Cet aspect est gênant. En effet, le programmeur aura naturellement tendance à écrire max avec deux arguments et non trois. Le programme ainsi obtenu ne rentrera pas dans la caractérisation. Du point de vue de l'intentionnalité, la caractérisation est très faible. Beaucoup d'algorithmes "naturels" en sont absents et le résultat est difficile à utiliser en pratique.

Cependant et même si cet argument supplémentaire est un gros frein à l'intentionnalité, il peut aussi être vu de manière plus positive comme une sorte de "certificat de complexité" qui pourrait accompagner un bout de code mobile pour informer l'ordinateur distant qui va l'exécuter sur les choses à faire pour obtenir la terminaison par LMPO (et donc une borne sur la complexité de l'algorithme).

La deuxième remarque, beaucoup plus positive, porte sur la complexité implicite. En effet, l'application stricte des règles de réécriture définies ci-dessus peut conduire à une chaîne de réductions de longueur exponentielle. Ceci est dû à des calculs qui sont effectués plusieurs fois (par exemple, dans le calcul de $\mathtt{lcs}(\mathbf{a}(\mathbf{a}(\epsilon)), \mathbf{b}(\mathbf{b}(\epsilon)))$, le terme $\mathtt{lcs}(\mathbf{a}(\epsilon), \mathbf{b}(\epsilon))$ est calculé deux fois). D'un point de vue algorithmique, ceci se résout de manière très classique par la technique dite de *programmation dynamique* (voir par exemple le chapitre 16 de [CLR94]).

Ainsi, bien que l'algorithme soit de complexité explicite exponentielle, la terminaison par LMPO permet de détecter que la fonction, elle, est bien de complexité polynomiale. Ce n'est pas l'algorithme qui est analysé, mais bel et bien la fonction. On dispose ainsi d'une méthode d'analyse de la complexité *implicite* de l'algorithme.

Bien sûr, cette analyse de la complexité implicite ne serait pas très intéressante si il était impossible d'atteindre la borne obtenue. Fort heureusement, on peut facilement transformer le système en un autre qui, lui, calcule bien en temps polynômial. Il faut pour cela utiliser la technique de *mémoïsation* introduite par S. Cook [Coo71] et utilisée par N. Andersen et N. D. Jones [AJ94, Jon97] ou A. Liu et S. D. Stoller [LS99].

L'idée est la suivante : on stocke le résultat de chaque appel de fonction dans un cache. Quand un nouvel appel est fait, on commence par regarder dans le cache. Si l'appel à faire s'y trouve déjà, on n'a pas besoin de refaire le calcul car on connaît déjà le résultat. Sinon, on fait le calcul et on stocke dans le cache la valeur obtenue, en vue d'une réutilisation. Ceci est obtenu en modifiant la sémantique opérationnelle des systèmes de réécriture. On dispose alors d'un interpréteur qui fonctionne comme décrit à la figure 4.1. Le cache ainsi utilisé est similaire au "Minimal Function Graph" de N. D. Jones [Jon97].

Il est aussi très intéressant de noter que, dans le cadre général, la mémoïsation n'est pas vraiment utilisable car il est impossible de savoir si un résultat sera réutilisé ou non. Ici, l'ordre de terminaison d'une part garantit la complexité polynomiale du résultat et d'autre part donne suffisamment d'informations sur le système de réécriture pour ne pas stocker de données inutiles dans le cache (voir [Mar00b] à ce sujet).

4.2 Quasi-interprétations

Les quasi-interprétations interprètent chaque symbole par une fonction bornée par un polynôme. Contrairement aux interprétations polynomiales, leur croissance n'est pas nécessairement stricte et la décroissance qu'elles imposent aux termes d'une réduction n'est pas forcément stricte non plus. Du coup, elles ne suffisent plus à prouver la terminaison d'un système de réécriture mais permettent de borner la taille des termes utilisés. Combinées avec des ordres de terminaison, elles permettent d'obtenir le même résultat que les ordres allégés.

4.2.1 Quasi-interprétations

Les quasi-interprétations ont été introduites par J.-Y. Marion et J.-Y. Moyen [MM00] et nous les avons réutilisées avec G. Bonfante [BMM01]. R. Amadio a montré qu'on peut décider si un système

Étant donné un ensemble de règles $\mathcal{E}$ et une substitution σ, $t \xrightarrow{!} v$ en utilisant le cache C et en obtenant le cache C' s'écrit :

$$\mathcal{E}, \sigma \vdash \langle C, t\rangle \to \langle C', v\rangle$$

En particulier,

$$\mathcal{E}, \emptyset \vdash \langle \emptyset, \mathtt{f}(t_1, \ldots, t_n)\rangle \to \langle C, v\rangle \text{ signifie } [\![\mathtt{f}]\!](t_1, \ldots, t_n) = v$$

$$(1) \quad \frac{\sigma(x) = v}{\mathcal{E}, \sigma \vdash \langle C, x\rangle \to \langle C, v\rangle} \text{(Variable)}$$

$$(2) \quad \frac{\mathbf{c} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathbf{c}(t_1, \ldots, t_n)\rangle \to \langle C_n, \mathbf{c}(v_1, \ldots, v_n)\rangle} \text{(Constructeur)}$$

$$(3) \quad \frac{\mathtt{f} \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle \quad (\mathtt{f}(v_1, \ldots, v_n), v) \in C_n}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \ldots, t_n)\rangle \to \langle C_n, v\rangle} \text{(Lecture dans le cache)}$$

$$(4) \quad \frac{\mathtt{f} \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle \quad \mathtt{f}(p_1, \ldots, p_n) \to r \in \mathcal{E} \quad p_i\sigma' = v_i \quad \mathcal{E}, \sigma' \vdash \langle C_n, r\rangle \to \langle C, v\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \ldots, t_n)\rangle \to \langle C \bigcup (\mathtt{f}(v_1, \ldots, v_n), v), v\rangle}$$

(Mise en cache)

FIG. 4.1 – Évaluation d'un système de réécriture avec mise en cache des résultats intermédiaires.

de réécriture admet une quasi-interprétation qui est un polynôme sur l'algèbre $(\max, +)$ [Ama02], ce problème étant NPTIME-complet.

Définition 70 (Quasi-interprétations polynômiales).
Une *quasi-interprétation polynomiale* d'un symbole $a \in \mathcal{F} \bigcup \mathcal{C}$ d'arité n est une fonction $(\!|a|\!) : \mathbb{N}^n \to \mathbb{N}$ telle que :
- $(\!|a|\!)$ est bornée par un polynôme.
- $(\!|a|\!)(X_1, \ldots, X_n) \geq X_i$ pour tout $1 \leq i \leq n$.
- $(\!|a|\!)$ est croissante (non-strictement) vis-à-vis de chacune de ses variables.
- $(\!|\mathbf{c}|\!)(X_1, \ldots, X_n) = \sum_{i=1}^{n} X_i + b$ si $\mathbf{c} \in \mathcal{C}$ (b est une constante strictement positive).

La quasi-interprétation d'une constante (constructeur d'arité 0) est donc une constante strictement positive.

Les quasi-interprétations sont étendues aux termes clos de manière canonique :

$$(\!|\mathtt{f}(t_1, \ldots, t_n)|\!) = (\!|\mathtt{f}|\!)((\!|t_1|\!), \ldots, (\!|t_n|\!))$$

On peut remarquer que la condition sur les constructeurs est d'avoir une quasi-interprétation de genre 0 (en reprenant la terminologie de G. Bonfante, A. Cichon, J.-Y. Marion et H. Touzet). C'est très important pour obtenir des bornes polynomiales au final.

Définition 71.
Un système de réécriture *admet* une quasi-interprétation $(\!|\bullet|\!)$ si pour chaque règle $l \to r$ et pour chaque substitution close σ, $l\sigma \geq r\sigma$.

Le fait qu'on autorise une décroissance non-stricte entre les deux côtés d'une règle implique que les quasi-interprétations ne sont pas suffisantes pour prouver la terminaison du système. En effet, le système de réécriture très simple ne comportant que la règle

$$\mathtt{f}(x) \to \mathtt{f}(x)$$

ne termine pas mais admet une quasi-interprétation.

Cependant, les quasi-interprétations vont permettre de borner la taille des termes et, couplées avec d'autres méthodes de terminaison, d'obtenir ainsi une borne sur la complexité du système.

Lemme 24.
Si $s \trianglelefteq t$ *alors* $(\!|s|\!) \leq (\!|t|\!)$.

Démonstration.
$(\!|\mathtt{f}|\!)(X_1, \ldots, X_n) \geq X_i$ pour tout $1 \geq i \geq n$. □

Lemme 25.
Soient t *et* s *deux termes constructeurs et* $(\!|\bullet|\!)$ *une quasi-interprétation. Si* $s \triangleleft t$ *alors* $(\!|s|\!) < (\!|t|\!)$.

Démonstration.
Immédiat d'après la forme des quasi-interprétations pour les constructeurs. □

Lemme 26.
Si $t \in \mathcal{T}(\mathcal{C})$ *alors* $|t| \leq (\!|t|\!)$ *et il existe une constante* c *telle que* $(\!|t|\!) \leq c \cdot |t|$.

Démonstration.
Par induction sur $|t|$. C'est vrai pour $|t| = 1$ parce que dans ce cas $t = \mathbf{c} \in \mathcal{C}$ donc $(\!|t|\!) = a \geq 1$.
Si $t = \mathbf{c}(t_1, \ldots, t_n)$ alors $(\!|t|\!) = \sum_{i=1}^{n} (\!|t_i|\!) + a \geq \sum_{i=1}^{n} |t_i| + 1 = |t|$.

De même, si $|t| = 1$, $(\!|t|\!) = a \leq c$ (il suffit de prendre $c = \max(a)$) et sinon

$$(\!|t|\!) = \sum_{i=1}^{n} (\!|t_i|\!) + a \leq \sum_{i=1}^{n} c \cdot |t_i| + a \leq c \cdot (\sum_{i=1}^{n} |t_i| + 1)$$

□

Lemme 27.
Si $(\!|t|\!) \leq (\!|s|\!)$ *alors* $(\!|f(\cdots, t, \cdots)|\!) \leq (\!|f(\cdots, s, \cdots)|\!)$.

Démonstration.
Immédiat car les quasi-interprétations sont croissantes. □

Proposition 28.
Si $t \xrightarrow{*} s$ *alors* $(\!|s|\!) \leq (\!|t|\!)$.

Démonstration.
Un pas de réduction a lieu sur un rédex u de t. Donc il existe une règle $l \to r$ et une substitution σ telles que $l\sigma = u$. Par définition des quasi-interprétations d'un système de réécriture, $(\!|l\sigma|\!) \geq (\!|r\sigma|\!)$ donc par application du lemme ci-dessus, un pas de réduction ne fait jamais croître la quasi-interprétation. □

Corollaire 29.
Soit main un système de réécriture qui admet une quasi-interprétation polynomiale.

Pour chaque symbole de fonction f *et chaque terme constructeur* $v_0, \ldots, v_n$ *tels que* $f(v_1, \ldots, v_n) \xrightarrow{!} v_0$, $|v_0| \leq P(\max_i(|v_i|))$ *pour un polynôme* P.

Démonstration.
Par définition des quasi-interprétations, $(\!|\mathtt{f}(v_1, \ldots, v_n)|\!)$ est borné par un polynôme en $\max_i((\!|v_i|\!))$, donc par un polynôme en $\max_i(|v_i|)$ car la quasi-interprétation des termes constructeurs est polynomialement bornée par leur taille (lemme 26). De même, $|v_0| \leq (\!|v_0|\!)$. La proposition 28 permet de dire que $(\!|v_0|\!) \leq (\!|\mathtt{f}(v_1, \ldots, v_n)|\!)$ est donc bornée par un polynôme en $\max_i(|v_i|)$. □

4.2.2 Caractérisation de Ptime

Théorème 30 (J.-Y. Marion et J.-Y. Moyen, 2000 **[MM00]).**
La classe des systèmes de réécriture terminant par MPO' et admettant une quasi-interprétation caractérise Ptime.

C'est-à-dire que toute fonction de PTIME *est calculable par un système de réécriture qui termine par MPO' et admet une quasi-interprétation et que tout système de réécriture qui termine par MPO' et admet une quasi-interprétation calcule une fonction de* PTIME.

$$\boxed{\text{MPO' + Quasi-interprétations} \equiv \textsc{Ptime}.}$$

Exemple 12 (Encore `lcs`).
Le système de réécriture suivant calcule la longueur de la plus longue sous suite commune à deux suites :

$\mathtt{max} : \mathtt{Nat}, \mathtt{Nat} \to \mathtt{Nat}$

$$\begin{aligned} \mathtt{max}(n, \mathbf{0}) &\to n \\ \mathtt{max}(\mathbf{0}, m) &\to m \\ \mathtt{max}(\mathbf{S}(n), \mathbf{S}(m)) &\to \mathbf{S}(\mathtt{max}(n, m)) && \mathbf{i} \in \{\mathbf{a}, \mathbf{b}\} \end{aligned}$$

$\mathtt{main} = \mathtt{lcs} : \mathtt{Word}, \mathtt{Word} \to \mathtt{Word}$

$$\begin{aligned} \mathtt{lcs}(\epsilon, \epsilon) &\to \mathbf{0} \\ \mathtt{lcs}(\epsilon, y) &\to \mathbf{0} \\ \mathtt{lcs}(x, \epsilon) &\to \mathbf{0} \\ \mathtt{lcs}(\mathbf{i}(x), \mathbf{i}(y)) &\to \mathbf{S}(\mathtt{lcs}(x, y)) \\ \mathtt{lcs}(\mathbf{i}(x), \mathbf{j}(y)) &\to \mathtt{max}(\mathtt{lcs}(x, \mathbf{j}(y)), \mathtt{lcs}(\mathbf{i}(x), y)) && \mathbf{i} \neq \mathbf{j} \end{aligned}$$

Ce programme termine par MPO' en choisissant $\mathtt{max} \prec_{\mathcal{F}} \mathtt{lcs}$. Il admet la quasi-interprétation :

$$(\!|\mathbf{0}|\!) = (\!|\epsilon|\!) = 1 \quad (\!|\mathbf{S}|\!)(X) = (\!|\mathbf{i}|\!)(X) = (\!|\mathbf{j}|\!)(X) = X + 1$$

$$(\!|\mathtt{max}|\!)(X, Y) = \max(X, Y) \quad (\!|\mathtt{lcs}|\!)(X, Y) = \max(X, Y)$$

Donc le système de réécriture calcule une fonction de PTIME.

La première remarque que cet exemple appelle est que, comme pour la terminaison par LMPO, c'est bien la complexité *implicite* de l'algorithme qui est capturée et non sa complexité explicite. Là encore, pour atteindre la borne polynomiale, il va falloir simuler la programmation dynamique au moyen de la mémoïsation, c'est-à-dire utiliser l'interpréteur avec mise en cache de la figure 4.1.

La deuxième remarque est que l'argument inutile de `max` a été supprimé. Le système de réécriture analysé est celui qui serait écrit naturellement par un programmeur. Du point de vue de l'intentionnalité, cette caractérisation est nettement meilleure que LMPO puisqu'elle capture plus d'algorithmes naturels.

Cependant, pour déterminer la borne sur la complexité on a toujours besoin de quelque-chose de plus que le simple programme. C'est ici la quasi-interprétation qui joue le rôle de certificat de complexité de l'algorithme.

Finalement, on peut aussi remarquer que `lcs` n'admet *aucune* interprétation polynomiale. Les quasi-interprétations sont beaucoup plus souples que les interprétations et permettent ainsi d'obtenir plus facilement des bornes sur la complexité.

4.3 Preuve de la caractérisation

Contrairement à la preuve donnée dans [MM00] qui transforme les programmes terminant par MPO' en programmes terminant par LMPO, la preuve donnée ici est directe. La preuve directe s'inspire de la preuve donnée pour LMPO dans [Mar00a] en remarquant que la section 5 de cet article a pour but de construire un polynôme qui vérifie les théorèmes 29 et 30 et que les quasi-interprétations vérifient directement ces deux propriétés (il s'agit ici des théorèmes 28 et 29). La fin de la preuve reprend ensuite la section 6 de cet article. De même, la quasi-interprétation construite à partir de l'article de S. Bellantoni et S. Cook [BC92] est définie explicitement.

4.3.1 Ptime est dans "$MPO' + (\!|\bullet|\!)$".

Définition 72 (La classe B de S. Bellantoni et S. Cook).
Dans [BC92], S. Bellantoni et S. Cook définissent une classe B de systèmes de réécriture. Cette classe est la plus petite classe qui contient les constructeurs :
- (**Constante**) $\mathbf{0}$;
- (**Successeurs**) $\mathbf{S}_i(x)$ pour $i \in \{0,1\}$;

les fonctions initiales :
- (**Projection**) $\pi_j^{n,m}(x_1,\ldots,x_n;x_{n+1},\ldots,x_{n+m}) \to x_j$ pour $1 \leq j \leq n+m$;
- (**Prédécesseur**) $\mathtt{p}(;\mathbf{0}) \to \mathbf{0} \quad p(;\mathbf{S}_i(x)) \to x$
- (**Conditionnelle**) $\mathtt{C}(;\mathbf{0},x,y) \to x \quad \mathtt{C}(;\mathbf{S}_0,x,y) \to x \quad \mathtt{C}(;\mathbf{S}_1,x,y) \to y$;

et est close par :
- (**Récursion prédicative**)

$$\mathtt{f}(\mathbf{0},x_1,\ldots,x_n;y_1,\ldots,y_m) \to \mathtt{g}(x_1,\ldots,x_n;y_1,\ldots,y_m)$$

$$\mathtt{f}(\mathbf{S}_i(z),x_1,\ldots,x_n;y_1,\ldots,y_m) \to \mathtt{h}_i(z,x_1,\ldots,x_n;y_1,\ldots,y_m,\mathtt{f}(z,x_1,\ldots,x_n;y_1,\ldots,y_m))$$

  pour $i \in \{0,1\}$ avec $\mathtt{g}$ et $\mathtt{h}_i$ dans B (définies précédemment) ;
- (**Composition sûre**)

$$\mathtt{f}(x_1,\ldots,x_n;y_1,\ldots,y_m) \to \mathtt{g}(\mathtt{h}_1(x_1,\ldots,x_n),\ldots,\mathtt{h}_p(x_1,\ldots,x_n);$$

$$\mathtt{l}_1(x_1,\ldots,x_n;y_1,\ldots,y_m),\ldots,\mathtt{l}_q(x_1,\ldots,x_n;y_1,\ldots,y_m))$$

  avec $\mathtt{g},\mathtt{h}_i$ et $\mathtt{l}_j$ dans B (définies précédemment).

Il est facile de voir que toutes les règles autorisées font décroître MPO' (et même LMPO en fait en mettant de valence 1 tous les arguments avant les point-virgules et de valence 0 les autres).

Définition 73 (Une quasi-interprétation pour la classe B).
Le lemme 4.1 de ce même article fournit une quasi-interprétation pour les systèmes de réécriture de B :
- $(\!|\mathbf{0}|\!) = 1$;
- $(\!|\mathbf{S}_i|\!)(X) = X + 1$;
- $(\!|\pi|\!)(X_1,\ldots,X_{n+m}) = \max(X_1,\ldots,X_{n+m})$;
- $(\!|\mathtt{p}|\!)(X) = X$;
- $(\!|\mathtt{C}|\!)(X,Y,Z) = \max(X,Y,Z)$;

Pour les fonctions définies par récursion prédicative ou composition sûre, $(\!|\mathtt{f}|\!)(X_1,\ldots,X_n;Y_1,\ldots,Y_m) = q_{\mathtt{f}}(X_1,\ldots,X_n) + \max(Y_1,\ldots,Y_m)$ avec $q_{\mathtt{f}}$ définie récursivement :
- $q\mathtt{f}(A,X_1,\ldots,X_n) = A(q_{\mathtt{h}_0}(A,X_1,\ldots,X_n) + q_{\mathtt{h}_1}(A,X_1,\ldots,X_n)) + g_{\mathtt{g}}(X_1,\ldots,X_n)$ si $\mathtt{f}$ est définie par récursion prédicative ;
- $q_{\mathtt{f}}(X_1,\ldots,X_n) = q_{\mathtt{g}}(q_{\mathtt{h}_1}(X_1,\ldots,X_n),\ldots,q_{\mathtt{h}_p}(X_1,\ldots,X_n)) + \sum_i q_{\mathtt{l}_i}(X_1,\ldots,X_n)$ si $\mathtt{f}$ est définie par composition sûre.

Ainsi, toute fonction polynomiale est calculable par un système de réécriture de B qui termine par MPO' et admet une quasi interprétation.

□

4.3.2 "$MPO' + (\!|\bullet|\!)$" est dans Ptime.

Réciproquement, tout système de réécriture qui termine par MPO' et admet une quasi-interprétation est calculable en temps polynômial par l'interpréteur avec cache de la figure 4.1. Commençons par prouver que la taille du cache est bornée par un polynôme dans la taille des entrées.

Définition 74 (Rang d'un symbole de fonction).
La précédence permet de définir un rang sur les symboles de fonctions. Le *rang* d'un symbole est le plus petit entier p tel que tous les symbole inférieurs (pour la précédence) soient de rang au plus $p-1$.

C_p est alors l'ensemble des n-uplets de termes constructeurs qui sont arguments d'appels de fonctions de rang p. On a donc

$$|C| \leq \sum_{1 \leq p \leq k} |C_p|$$

où k est le plus grand rang des symboles de $\mathcal{F}$ et $|A|$ désigne le cardinal de l'ensemble A.

Borner la taille du cache.

Pour borner $|C_p|$, on le partitionne en deux ensembles $C_p^{\vee}$ et $C_p^{\wedge}$. Considérons la réduction de r qu'il faut faire après application d'une règle $l \to r$ avec une substitution σ (cas 4 de l'interpréteur). On a donc $l\sigma = \mathtt{f}(v_1, \ldots, v_m)$ avec $v_i \in \mathcal{T}(\mathcal{C})$. Soit $\mathtt{g}(u_1, \ldots, u_n) \lhd r\sigma$ où $\mathtt{g}$ est de rang p. Comme le système de réécriture termine par MPO', on a nécessairement $\mathtt{g}(u_1, \ldots, u_n) \prec_{mpo} \mathtt{f}(v_1, \ldots, v_m)$. Il y a donc deux cas possibles.

1. Si le rang de $\mathtt{f}$ est strictement plus grand que p ($\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$), alors on met $(u'_1, \ldots, u'_n)$ dans $C_p^{\vee}$ (u'_i est la forme normale de u_i). Par conséquent, le cardinal de $C_p^{\vee}$ est borné par

$$|C_p^{\vee}| \leq \sum_{i<p} |C_i|$$

2. Dans le cas contraire, $\mathtt{f}$ et $\mathtt{g}$ ont le même rang, donc $n = m$ et $(u_1, \ldots, u_n) \trianglelefteq^{m'} (v_1, \ldots, v_n)$. En fait, tous les appels de fonctions de rang p faits lors de la normalisation de $\mathtt{f}(v_1, \ldots, v_n)$ se font avec des arguments qui sont des sous-termes des v_i. Ils appartiennent donc à

$$I(v_1, \ldots, v_n) = \{(u_1, \ldots, u_n) : (u_1, \ldots, u_n) \trianglelefteq^{m'} (v_1, \ldots, v_n)\}$$

on peut donc borner le cardinal de $C_p^{\wedge}$ par

$$|C_p^{\wedge}| \leq |\mathcal{F}_p| \cdot \sum_{(v_1, \ldots, v_n) \in C_p^{\vee}} |I(v_1, \ldots, v_n)|$$

Par un simple argument de comptage, on peut voir que $|I(v_1, \ldots, v_n)| \leq n! \cdot \sum_{i \leq n} |v_i|$. De plus, le théorème 29 implique que pour chaque $(v_1, \ldots, v_n) \in C_p^{\vee}$, il existe un polynôme P tel que $\sum_{i \leq n} |v_i| \leq n \cdot P(\max_{i \leq r} |a_i|)$ où les a_i sont les entrées de l'algorithme. En combinant tous ces résultats, on obtient donc

$$|C_p^{\wedge}| \leq |C_p^{\vee}| \cdot P'(\max_{i \leq r} |a_i|)$$

où r est l'arité de `main` et $P'(X) = |\mathcal{F}_p| \cdot n! \cdot n \cdot P(X)$ est un polynôme (n est l'arité maximale d'un symbole).

On peut donc regrouper les bornes sur $C_p^{\vee}$ et $C_p^{\wedge}$ puisque

$$|C_p| \leq |C_p^{\vee}| + |C_p^{\wedge}|$$

Le cardinal de chacun des C_p est donc borné par un polynôme en la taille des entrées. Donc le cardinal de l'ensemble du cache est aussi borné par un tel polynôme (il n'y a qu'un nombre fini de C_p, au maximum autant que de symboles de fonctions). Comme les valeurs stockées dans le cache sont aussi bornées par un polynôme en la taille des entrées (c'est encore une conséquence du théorème 29), la taille du cache est bornée par un polynôme en la taille des entrées.

Pour conclure, il suffit de remarquer que le nombre de triplets (C, t, σ) obtenables lors d'une réduction $\mathcal{E}, \sigma \vdash \langle C, t \rangle \to \langle C', v \rangle$ est borné par un polynôme. En effet, $|C|$ est borné par un polynôme donc il existe un nombre polynômial de caches différents atteignables, t est un sous-terme d'une équation de $\mathcal{E}$ et il n'en existe donc qu'un nombre polynômial et σ assigne à chaque variable une valeur qui est un sous-terme d'une valeur de C.

Comme il n'y a qu'un nombre polynômial de configurations atteignables et que le système de réécriture termine (par MPO'), il termine forcément en temps polynômial.

$\square$

4.4 ICAR

Savoir si un système de réécriture termine par MPO (ou LPO) est décidable. C'est un problème qui est dans NPTIME si la précédence est inconnue (et dans PTIME si elle est connue).

Dans le cadre particulier que je présente ici (les symboles de fonctions et de constructeurs forment deux ensembles disjoints), la précédence est facile à trouver. Il suffit en effet de parcourir le programme et de construire un ensemble de contraintes de la forme $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$ si g apparaît dans le membre droit d'une règle de réécriture de f.

Savoir si le système admet une quasi-interprétation n'est peut-être pas décidable, mais si la quasi-interprétation est fournie, il est possible de vérifier qu'il s'agit bien d'une quasi-interprétation. De plus, comme les quasi-interprétations sont très liées à la taille des termes, le programmeur est normalement capable de fournir un candidat raisonnable au moment où il écrit le programme.

C'est sur ce principe que j'ai écrit le système ICAR (Implicit Complexity AnalyseR) [Moy01]. Le programme prend un système de réécriture et un candidat pour être quasi-interprétation et vérifie si le système termine par MPO' (ou LPO) et si le candidat est bien une quasi-interprétation. Voici un exemple de session avec ICAR :
Commençons par charger un programme et l'imprimer.

```
>load mult
>print
add(Z,x0) -> x0
add(S(x0),x1) -> S(add(x0,x1))
mult(Z,x0) -> Z
mult(S(x0),x1) -> add(x1,mult(x0,x1))

[S](x0)=(x0)+(1)
[Z]()=1
[mult](x0,x1)=(x0)*(x1)
[add](x0,x1)=(x0)+(x1)
```

On remarque que le programme contient à la fois les règles de réécriture et la quasi-interprétation potentielle. ICAR peut maintenant vérifier la complexité du programme chargé.

```
>check
The program terminates by MPO.
Quasi-interpretation are OK.
The function is Ptime computable.
```

Quand un programme est chargé, on peut normaliser des termes à l'aide de l'interpréteur "normal" ou à l'aide de l'interpréteur avec mise en cache.

```
>load lcs
>print
maxi(Z,x0) -> x0
maxi(x0,Z) -> x0
maxi(S(x0),S(x1)) -> S(maxi(x0,x1))
lcs(E,x0) -> Z
lcs(x0,E) -> Z
lcs(A(x0),A(x1)) -> S(lcs(x0,x1))
lcs(B(x0),B(x1)) -> S(lcs(x0,x1))
lcs(A(x0),B(x1)) -> maxi(lcs(A(x0),x1),lcs(x0,B(x1)))
lcs(B(x0),A(x1)) -> maxi(lcs(B(x0),x1),lcs(x0,A(x1)))

[B](x0)=(x0)+(1)
[A](x0)=(x0)+(1)
[E]()=1
[S](x0)=(x0)+(1)
[Z]()=1
```

```
[lcs](x0,x1)=max(x1,x0)
[maxi](x0,x1)=max(x1,x0)

>check
The program terminates by MPO.
Quasi-interpretation are OK.
The function is Ptime computable.

>setcbv
>getsem
Call by value
>eval lcs(A(A(A(E))),B(B(B(E))))
Time usage : 219
Result     : Z

>setmem
>getsem
Memoization
>eval lcs(A(A(A(E))),B(B(B(E))))
Time usage : 97
Result     : Z
```

`setcbv` et `setmem` choisissent l'interpréteur normal (appel par valeur) ou avec mémoïsation. `getsem` donne le nom de l'interpréteur utilisé actuellement.

La mesure de temps et correspond au nombre de pas de réductions effectués lors de l'exécution.

Quelques calculs de normalisation de $\mathtt{lcs}(\mathbf{a}^n(\epsilon), \mathbf{b}^n(\epsilon))$ donnent les temps de calcul suivants :

n	Appel par valeurs	Mémoïsation
0	4	4
1	17	17
2	63	48
3	219	97
4	771	164
5	2775	249
6	10169	352

On peut remarquer que le temps de calcul avec mémoïsation est effectivement quadratique.

Chapitre 5

Une caractérisation de PSPACE

Même si PSPACE est une classe sensiblement plus grande que PTIME, et en particulier n'est pas considérée comme regroupant des algorithmes efficaces, il peut être intéressant de la caractériser. En effet, plusieurs problèmes naturels sont dans PSPACE, comme par exemple la validité des formules booléennes quantifiées, savoir si une expression régulière accepte tous les mots possibles ou savoir si un programme booléen renvoie la réponse **True**. De plus, C. S. Lee, N. D. Jones et A. M. Ben Amram ont montré que leur algorithme de "Size-Change Principle" [LJBA01] pour détecter la terminaison de programmes est dans PSPACE.

Des caractérisations de PSPACE par récursion bornée se retrouvent chez A. Cobham [Cob62] et D. B. Thompson [Tho72]. Plus récemment, N. D. Jones a montré qu'une certaine classe de programmes fonctionnels (récursifs terminaux et ne pouvant pas écrire) restreint au premier ordre caractérise PSPACE. Là aussi, les ordres supérieurs caractérisent E_kSPACE. Cette caractérisation ressemblant à celles de V. Sazonov [Saz80], Y. Gurevich [Gur83] ou A. Goerdt [Goe92].

Sur le modèle de LMPO, A. Cichon et J.-Y. Marion ont introduit l'analyse prédicative dans LPO grâce aux valences. Ils ont ainsi construit le Light Lexicographic Path Ordering (LLPO) qui impose à LPO une décroissance sur un argument de valence 1. Ils ont montré que LLPO caractérise PSPACE.

5.1 Caractérisation de PSPACE

De manière similaire à LMPO, on peut remplacer les valences par des quasi-interprétations polynomiales. On obtient donc la caractérisation suivante :

Théorème 31 (G. Bonfante, J.-Y. Marion et J.-Y. Moyen, 2001 **[BMM01]).**
La classe des systèmes de réécriture terminant par LPO et admettant une quasi-interprétation polynomiale caractérise PSPACE.

C'est-à-dire que toute fonction de PSPACE *est calculable par un système de réécriture qui termine par LPO et admet une quasi-interprétation polynomiale et que tout système de réécriture qui termine par LPO et admet une quasi-interprétation polynomiale calcule une fonction de* PSPACE.

$$\boxed{\text{LPO + Quasi-interprétations} \equiv \text{PSPACE.}}$$

Exemple 13 (Quantified Boolean Formula).
Le programme suivant calcule si une formule booléenne quantifiée est satisfiable (les formules sont restreintes à $\neq, \vee, \exists$ qui sont suffisants pour tout écrire) :

$$\begin{array}{rlrl}
\mathtt{not}(\mathbf{True}) & \to \mathbf{False} & \mathbf{0} = \mathbf{0} & \to \mathbf{True} \\
\mathtt{not}(\mathbf{False}) & \to \mathbf{True} & \mathbf{S}(x) = \mathbf{0} & \to \mathbf{False} \\
\mathtt{or}(\mathbf{True}, x) & \to \mathbf{True} & \mathbf{0} = \mathbf{S}(y) & \to \mathbf{False} \\
\mathtt{or}(\mathbf{False}, x) & \to x & \mathbf{S}(x) = \mathbf{S}(y) & \to x = y
\end{array}$$

$$\begin{array}{rl}
\mathtt{main}(\phi) & \to \mathtt{ver}(\phi, \mathbf{Nil}) \\
\mathtt{in}(x, \mathbf{Nil}) & \to \mathbf{False} \\
\mathtt{in}(x, \mathbf{Cons}(a, l)) & \to \mathtt{or}(x = a, \mathtt{in}(x, l))
\end{array}$$

Dans la fonction suivante, t est l'ensemble des variables dont la valeur est **True**.

$$\begin{array}{rl}
\texttt{ver}(\mathbf{Var}(x), t) & \rightarrow \texttt{in}(x, t) \\
\texttt{ver}(\mathbf{Or}(\phi_1, \phi_2), t) & \rightarrow \texttt{or}(\texttt{ver}(\phi_1, t), \texttt{ver}(\phi_2, t)) \\
\texttt{ver}(\mathbf{Not}(\phi), t) & \rightarrow \texttt{not}(\texttt{ver}(\phi, t)) \\
\texttt{ver}(\mathbf{Exists}(n, \phi), t) & \rightarrow \texttt{or}(\texttt{ver}(\phi, \mathbf{Cons}(n, t)), \texttt{ver}(\phi, t))
\end{array}$$

Ces règles sont ordonnées par LPO avec la précédence $\{\texttt{not}, \texttt{or}, \texttt{\_=\_}\} \prec_{\mathcal{F}} \texttt{in} \prec_{\mathcal{F}} \texttt{ver} \prec_{\mathcal{F}} \texttt{main}$.
Elle admettent la quasi-interprétation polynomiale suivante :

– $(\!|\mathbf{c}|\!)(X_1, \ldots, X_n) = 1 + \sum_{i=1}^{n} X_i$, pour chaque constructeur n-aire.
– $(\!|\texttt{ver}|\!)(\Phi, T) = \Phi + T$, $(\!|\texttt{main}|\!)(\Phi) = \Phi + 1$.
– $(\!|\texttt{f}|\!)(X_1, \ldots, X_n) = \max_{i=1}^{n} X_i$, pour les autres symboles de fonction.

La fonction est donc calculable en espace polynômial.

Là aussi, la quasi-interprétation peut jouer le rôle de certificat de complexité accompagnant du code mobile.

5.2 Preuve du théorème 31

5.2.1 "$LPO + (\!|\bullet|\!)$" est dans PSPACE.

Pour obtenir ce résultat, il ne faut pas normaliser le terme sans faire attention. En effet, voici un programme

$$\begin{array}{rcl}
\texttt{f}(\mathbf{0}, y) & \rightarrow & \texttt{max}(y, y) \\
\texttt{f}(\mathbf{S}(x), y) & \rightarrow & \texttt{f}(x, \texttt{max}(y, y))
\end{array}$$

qui termine par LPO et admet une quasi interprétation (`max` est définit de la même manière que pour l'exemple de `lcs`). Cependant, le choix des rédex à chaque étape est très important pour l'espace utilisé. Si on choisit de réduire prioritairement les rédex qui commencent par `f` (stratégie "outermost" en terme de réécriture, évaluation "call by name" en terme de programmation), on aboutira à des termes de taille exponentielle (avec plusieurs `max` imbriqués). En revanche, si on choisit de réduire prioritairement les rédex qui commencent par `max` (stratégie "innermost" ou évaluation "call by value"), le calcul aura bien lieu en espace polynômial.

Arbres d'évaluation.

Un état représente un des appels récursifs qui peut être fait lors de l'évaluation avec appel par valeurs d'un terme.

Définition 75 (États).
Un *état* est un n-uplet $\langle \texttt{f}, t_1, \ldots, t_n \rangle$ où `f` est un symbole de fonction d'arité n et $t_1, \ldots, t_n$ sont des termes constructeurs.
$State(\texttt{main})$ est l'ensemble de tous les états construits à partir des symboles d'un programme `main`.
$State^A(\texttt{main}) = \{\langle \texttt{f}, t_1, \ldots, t_n \rangle \in State(\texttt{main}) \ / \ |t_i| < A\}$ est l'ensemble des états dont les sous-termes sont de taille bornée par A.

Définition 76 (Transition).
Soient *main* un programme, $\eta_1 = \langle \texttt{f}, t_1, \ldots, t_n \rangle$ et $\eta_2 = \langle \texttt{g}, s_1, \ldots, s_m \rangle$ deux états de $State(\texttt{main})$. Une *transition* est un triplet $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ tel que :

1. $e \equiv \texttt{f}(p_1, \ldots, p_n) \rightarrow t$.
2. Il existe une substitution σ telle que $p_i\sigma = t_i$ pour tout $1 \leq i \leq n$.
3. Il existe un sous-terme $\texttt{g}(u_1, \ldots, u_m) \trianglelefteq t$ tel que $u_i\sigma \overset{!}{\rightarrow} s_i$ pour tout $1 \leq i \leq n$.

$Transition(\texttt{main})$ est l'ensemble de toutes les transitions entre les éléments de $State(\texttt{main})$.
$\overset{*}{\rightsquigarrow}$ est la clôture réflexive transitive de $\cup_{e \in \mathcal{E}} \overset{e}{\rightsquigarrow}$.

Définition 77 (Arbre d'évaluation).
Soient *main* un système de réécriture qui termine par LPO et $\langle \mathtt{f}, t_1, \ldots, t_n \rangle$ un état (de $State(\mathtt{main})$). Un *arbre d'évaluation* Ω de $\langle \mathtt{f}, t_1, \ldots, t_n \rangle$ est défini récursivement :

- la racine de Ω est $\langle \mathtt{f}, t_1, \ldots, t_n \rangle$.
- Les enfants de chaque nœud η_1, sont les éléments de $\{\eta_2 \in State(\mathtt{main}) \ / \ \eta_1 \overset{e}{\rightsquigarrow} \eta_2 \in Transition(\mathtt{main})\}$ où $e \in \mathcal{E}$.

$CT(\langle \mathtt{f}, t_1, \ldots, t_n \rangle)$ est l'ensemble de tous les arbres d'évaluation de $\langle \mathtt{f}, t_1, \ldots, t_n \rangle$.

$$CT^A(\langle \mathtt{f}, t_1, \ldots, t_n \rangle) = \{\tau \in CT(\langle \mathtt{f}, t_1, \ldots, t_n \rangle) \ / \ \forall \eta \in \tau, \eta \in State^A(\mathtt{main})\}$$

est l'ensemble des arbres ne contenant que des états de taille bornée.

Exemple 14.
Si on reprend le système de réécriture pour calculer la fonction d'Ackermann, le seul arbre d'évaluation de $CT(\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{S}(\mathbf{0}) \rangle)$ est :

$$\begin{tikzcd}
& \boxed{\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{S}(\mathbf{0}) \rangle} \arrow[dl] \arrow[dr] & \\
\boxed{\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{0} \rangle} \arrow[d] & & \boxed{\langle \mathtt{Ack}, \mathbf{0}, \mathbf{S}(\mathbf{S}(\mathbf{0})) \rangle} \\
\boxed{\langle \mathtt{Ack}, \mathbf{0}, \mathbf{S}(\mathbf{0}) \rangle} & &
\end{tikzcd}$$

Borner la hauteur des arbres d'évaluation.

Lemme 32.
Soient main un système de réécriture qui termine par LPO, α le rang maximum d'un symbole de fonction, d l'arité maximale d'un symbole et A un entier (qui borne la taille des états de l'arbre d'évaluation). Pour tout arbre d'évaluation $\Omega \in CT^A(\langle f, t_1, \ldots, t_n \rangle)$, on a :

1. *Si $\langle f, t_1, \ldots, t_n \rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \ldots, s_m \rangle$ alors (a) $g \prec_{\mathcal{F}} f$ ou (b) $g \approx_{\mathcal{F}} f$ et $(s_1, \ldots, s_m) \prec^l_{lpo} (t_1, \ldots, t_n)$.*
2. *Si $\langle f, t_1, \ldots, t_n \rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \ldots, s_m \rangle \in \Omega$ et $g \approx_{\mathcal{F}} f$ alors le nombre d'états entre $\langle f, t_1, \ldots, t_n \rangle$ et $\langle g, s_1, \ldots, s_m \rangle$ est borné par A^d.*
3. *La longueur de chaque branche de Ω est bornée par $\alpha \times A^d$.*

Démonstration.

1. Parce que le programme termine par LPO, donc chaque règle fait décroître LPO.
2. Soit $\langle \mathtt{h}, u_1, \ldots, u_p \rangle$ un enfant de $\langle \mathtt{f}, t_1, \ldots, t_n \rangle$. Comme t_i et u_j sont des termes constructeurs, il existe une permutation telle que $(|u_1|, \cdots, |u_p|) <^l (|t_{\pi(1)}|, \cdots, |t_{\pi(p)}|)$. Comme il y a au plus A^d d-uplets dont chaque composante est bornée par A, la longueur de la chaîne décroissante est bornée par A^d.
3. Dans chaque branche, il y a au maximum A^d états avec des symboles de fonction ayant un même rang, puis A^d avec un rang immédiatement inférieur, *etc.* Comme il n'y a que α rangs différents, la longueur d'une branche est bornée par $\alpha \times A^d$.

□

Lemme 33.
Soient main un système de réécriture qui termine par LPO et admet une quasi-interprétation, f un symbole de fonction et $t_1, \ldots, t_n$ des termes constructeurs.

$$CT(f, t_1, \ldots, t_n) = CT^{(\!| f(t_1, \ldots, t_n) |\!)}(f, t_1, \ldots, t_n)$$

.

Démonstration.
Soient $\Omega \in CT(\mathtt{f}, t_1, \ldots, t_n)$ et $\langle \mathtt{g}, s_1, \ldots, s_m \rangle$ un état de Ω. Comme s_i est un terme constructeurs, $|s_i| \leq (\!| s_i |\!) \leq (\!| \mathtt{g}(s_1, \ldots, s_m) |\!) \leq (\!| \mathtt{f}(t_1, \ldots, t_n) |\!)$ par définition des quasi-interprétations. □

Lemme 34.
Soient $t_1, \dots, t_n$ des termes constructeurs. Il existe un polynôme P tel que

$$(\!|main(t_1, \dots, t_n)|\!) \leq P(\max_{i=1}^{n} |t_i|)$$

.

Démonstration.
Car $(\!|t_i|\!) \leq c \cdot |t_i|$. □

Appel par valeurs.

Pour obtenir la borne polynomiale sur l'espace utilisé, il suffit de calculer les termes de l'arbre d'évaluation dans l'ordre obtenu par un parcours en profondeur. Cela revient à appliquer une stratégie d'appel par valeurs (ou "innermost" en terme de réécriture) pour évaluer un terme.

Théorème 35.
Soient main un système de réécriture qui termine par LPO et admet une quasi-interprétation et $t_1, \dots, t_n$ des termes constructeurs. $\mathtt{main}(t_1, \dots, t_n)$ peut être réduit en espace polynômial à l'aide de la stratégie suivante :
À chaque étape, on choisit un rédex dont tous les sous-termes sont des termes constructeurs et on applique le nombre d'étapes nécessaire pour le normaliser.

Démonstration.
Cette stratégie revient à explorer entièrement une des branches de l'arbre d'évaluation d'un terme avant de passer à la suivante. La taille des termes obtenus lors de la réduction est donc bornée par la taille d'une branche plus la taille du contexte dans lequel il faut replacer cette branche. La taille des branches est bornée polynomialement par la taille des t_i (car le programme admet une quasi-interprétation). La taille du contexte est bornée par la taille d'un terme de l'arbre et l'arité maximale d'un nœud de l'arbre.

Donc la taille d'un terme obtenu pendant la réduction est borné polynomialement par la taille des termes initiaux. La fonction est bien calculable en espace polynômial. □

5.2.2 PSPACE est dans "$LPO + (\!|\bullet|\!)$".

Pour montrer que LPO et les quasi-interprétations permettent bien de calculer toutes les fonctions de PSPACE, je montre que les systèmes de réécriture terminant par LPO et admettant une quasi-interprétation permettent de simuler le modèle de calculs de "Parallel Register Machines" introduit par D. Leivant et J.-Y. Marion [LM95] qui sont une relecture des machines de Turing alternantes de A. Chandra, D. Kozen et L. Stockmeyer [CKS81] et qui capturent PSPACE.

Parallel Register Machines.

Définition 78 (PRM).
Une *Machine Parallèle à Registres* ("Parallel Register Machine" (PRM)) sur les mots binaires est composée de :

1. Un ensemble fini d'*états* $S = \{s_0, s_1, \dots, s_k\}$ qui contient notamment l'état BEGIN.
2. Une liste finie de *registres* $\Pi = \{\pi_1, \dots, \pi_m\}$; le dernier registre, π_m est aussi noté OUTPUT ; les valeurs des registres sont des mots sur l'alphabet $\{\mathbf{0}, \mathbf{1}\}$.
3. L'ordre lexicographique $<^l$ sur $\mathbb{W} = \{\mathbf{0}, \mathbf{1}\}^*$: $\epsilon <^l y$, $\mathbf{0}(x) <^l \mathbf{1}(y)$, $\mathbf{i}(x) <^l \mathbf{i}(y)$ si et seulement si $x <^l y$.
4. Une fonction *com* des états vers les *commandes* qui sont : $[\mathbf{Succ}(\pi' = \mathbf{i}(\pi), s')]$, $[\mathbf{Pred}(\pi' = \mathbf{p}(\pi), s')]$, $[\mathbf{Branch}(\pi, s', s'')]$, $[\mathbf{Fork}_{\min}(s', s'')]$, $[\mathbf{Fork}_{\max}(s', s'')]$, $[\mathbf{End}]$.

Définition 79 (Configuration).
Une *configuration* d'une PRM M est une paire (s, F) où $s \in S$ est un état et F est une substitution de Π vers $\mathbb{W}$. $[u_1, \dots, u_m]$ représente la fonction qui assigne u_i à π_i et $\{\pi_i \leftarrow a\}[u_1, \dots, u_m]$ représente $[u_1, \dots, u_{i-1}, a, u_{i+1}, \dots, u_m]$.

Définition 80 (Sémantique des PRMs).
Soit M une PRM. La fonction partielle de sémantique $\mathtt{eval} : \mathbb{N} \times S \times \mathbb{W}^m \mapsto \mathbb{W}$, donne le résultat du calcul de M étant donné une "horloge" représentée par le premier argument.
- $\mathtt{eval}(0, s, F)$ est non défini.
- Si $com(s) = \mathbf{Succ}(\pi' = \mathbf{i}(\pi), s')$ alors $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', \{\pi' \leftarrow \mathbf{i}(\pi)\}F)$ (à droite de la flèche, π représente le contenu du registre).
- Si $com(s) = \mathbf{Pred}(\pi' = \mathbf{p}(\pi), s')]$, alors $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', \{\pi' \leftarrow \mathbf{p}(\pi)\}F)$ où $\mathbf{p}(\epsilon) = \epsilon$ et $\mathbf{p}(\mathbf{i}(x)) = x$.
- Si $com(s) = \mathbf{Branch}(\pi, s', s'')$ alors $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, r, F)$, où $r = s'$ si $\pi = \mathbf{0}(w)$ et $r = s''$ si $\pi = \mathbf{1}(w)$.
- Si $com(s) = \mathbf{Fork}_{\min}(s', s'')$
  alors $\mathtt{eval}(t+1, s, F) = \min_{<^l}(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$.
- Si $com(s) = \mathbf{Fork}_{\max}(s', s'')$
  alors $\mathtt{eval}(t+1, s, F) = \max_{<^l}(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$.
- Si $com(s) = \mathbf{End}$ alors $\mathtt{eval}(t+1, s, F) = F(\text{OUTPUT})$.

Définition 81 (Calcul).
Soit T une fonction de $\mathbb{N}$ vers $\mathbb{N}$. La PRM M *calcule* $f : \mathbb{W}^k \to \mathbb{W}$ *en temps* T si pour tout $(w_1, \ldots, w_k) \in \mathbb{W}^k$, on a :

$$\mathtt{eval}(T(\max_{i=1}^{k} |w_i|), \text{BEGIN}, [w_1, \ldots, w_k, \epsilon, \ldots, \epsilon]) = f(w_1, \ldots, w_k)$$

Théorème 36 (A. Chandra, D. Kozen et L. Stockmeyer, 1981 **[CKS81]).**
Soit $f : \mathbb{W} \to \mathbb{W}$. f est calculable en espace polynômial si et seulement si il existe une PRM qui calcule f en temps polynômial.

Simulation des PRMs par des systèmes de réécriture.

La simulation est faite de manière très directe en reprenant chacune des règles pour le calcul de `eval` et en la transformant en une règle de réécriture.

Lemme 37 (Lemme Plug and Play).
Soit $f : \mathbb{W} \to \mathbb{W}$ une fonction calculable par une PRM M en temps T. La fonction f' suivante est calculable par un système de réécriture qui termine par LPO et admet une quasi-interprétation :

$$\begin{array}{rcll} f' : \mathbb{N} \times \mathbb{W} & \to & \mathbb{W} & \\ (n, w) & \mapsto & f(w) & \textit{si } n > T(|w|) \\ (n, w) & \mapsto & \perp & \textit{sinon} \end{array}$$

Démonstration.
Le système de réécriture a pour ensemble de constructeurs $\mathcal{C} = \{\mathbf{0}, \mathbf{1}, \mathbf{s}, \diamond, \epsilon\} \bigcup S$ où S est l'ensemble des états de M. Les symboles de fonctions sont : $\mathtt{min}, \mathtt{max}$ qui correspondent à $\min_{<^l}, \max_{<^l}$, `eval` qui simule les règles de la sémantique des PRMs et $\mathtt{f}'$. Les règles de réécriture sont les suivantes :

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{min}(\epsilon, w) & \to & \epsilon & \mathtt{max}(\epsilon, w) & \to & w \\ \mathtt{min}(w, \epsilon) & \to & \epsilon & \mathtt{max}(w, \epsilon) & \to & w \\ \mathtt{min}(\mathbf{0}(w), \mathbf{1}(w')) & \to & \mathbf{0}(w) & \mathtt{max}(\mathbf{0}(w), \mathbf{1}(w')) & \to & \mathbf{1}(w') \\ \mathtt{min}(\mathbf{1}(w), \mathbf{0}(w')) & \to & \mathbf{0}(w') & \mathtt{max}(\mathbf{1}(w), \mathbf{0}(w')) & \to & \mathbf{1}(w) \\ \mathtt{min}(\mathbf{i}(w), \mathbf{i}(w')) & \to & \mathbf{i}(\mathtt{min}(w, w')) & \mathtt{max}(\mathbf{i}(w), \mathbf{i}(w')) & \to & \mathbf{i}(\mathtt{max}(w, w')) \end{array}$$

avec $\mathbf{i} \in \{\mathbf{0}, \mathbf{1}\}$. On a $[\mathtt{min}] = \min_{<^l}$ et $[\mathtt{max}] = \max_{<^l}$.

1. $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \ldots, \pi_{j-1}, \mathbf{i}(\pi_k), \pi_{j+1}, \ldots, \pi_m)$
   si $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_k), s')$,

2. $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \mathbf{i}(\pi'_j), \ldots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \ldots, \pi'_j, \ldots, \mathbf{i}(\pi'_j), \ldots, \pi_m)$
   si $com(s) = \mathbf{Pred}(\pi_k = \mathbf{p}(\pi_j), s')$,

3. $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_{j-1}, \mathbf{i}(\pi_j), \pi_{j+1}, \ldots, \pi_m) \to \texttt{Eval}(t, r, \pi_1, \ldots, \pi_m)$
   si $com(s) = \mathbf{Branch}(\pi_j, s', s'')$ où $r = s'$ si $\mathbf{i} = \mathbf{0}$ et $r = s''$ si $\mathbf{i} = \mathbf{1}$,

4. $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \texttt{min}(\texttt{Eval}(t, s', \pi_1, \ldots, \pi_m), \texttt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
   si $com(s) = \mathbf{Fork}_{\min}(s', s'')$

5. $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \texttt{max}(\texttt{Eval}(t, s', \pi_1, \ldots, \pi_m), \texttt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
   si $com(s) = \mathbf{Fork}_{\max}(s', s'')$

6. $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \pi_m$
   si $com(s) = \mathbf{End}$

Avec la précédence $\{\texttt{min}, \texttt{max}\} \prec_{\mathcal{F}} \texttt{Eval}$, ces règles font décroître LPO parce que l'horloge (le premier argument) décroît. Le système admet la quasi-interprétation suivante :

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
(\!|\epsilon|\!) & = & 1 & (\!|\mathbf{0}|\!)(X) & = & X+1 & (\!|\texttt{min}|\!)(W, W') & = & \max(W, W') \\
(\!|\diamond|\!) & = & 1 & (\!|\mathbf{1}|\!)(X) & = & X+1 & (\!|\texttt{max}|\!)(W, W') & = & \max(W, W') \\
 & & & (\!|\mathbf{s}|\!)(X) & = & X+1 & & &
\end{array}$$

$$\forall s \in S, (\!|s|\!) = 1 \qquad (\!|\texttt{Eval}|\!)(T, S, \Pi_1, \ldots, \Pi_m) = \max\{\Pi_1, \ldots, \Pi_m\} + T + S$$

Il ne reste plus qu'à écrire la fonction $\texttt{f}'$ définie par $\texttt{f}'(n, w) \to \texttt{Eval}(n, \textsc{begin}, w, \epsilon, \ldots, \epsilon)$. Il est aisé de voir que $f' = [\texttt{f}']$. Cette règle décroît par LPO en prenant $\texttt{Eval} \prec_{\mathcal{F}} \texttt{f}'$ et elle admet la quasi-interprétation $(\!|\texttt{f}'|\!)(N, X) = N + X + 1$. □

Lemme 38.
Soient $w \in \mathbb{W}$ et P un polynôme, la fonction $w \in \mathbb{W} \mapsto P(|w|)$ est calculable par un système de réécriture terminant par LPO et admettant une quasi-interprétation.

Démonstration.
Tout polynôme peut être calculé par une combinaison de $\texttt{add}$ et $\texttt{mult}$.

$$\begin{array}{rcl@{\qquad}rcl}
\texttt{add}(\diamond, y) & \to & y & \texttt{mult}(\diamond, y) & \to & \diamond \\
\texttt{add}(\mathbf{s}(x), y) & \to & \mathbf{s}(\texttt{add}(x, y)) & \texttt{mult}(\mathbf{s}(x), y) & \to & \texttt{add}(y, \texttt{mult}(x, y))
\end{array}$$

Ces règles sont ordonnées par LPO avec la précédence $\texttt{add} \prec_{\mathcal{F}} \texttt{mult}$. Elle admettent la quasi-interprétation :

$$(\!|\texttt{add}|\!)(X, Y) = X + Y$$
$$(\!|\texttt{mult}|\!)(X, Y) = X \times Y$$

De même, la taille d'un terme peut être calculée à l'aide de la fonction

$$\texttt{size}(\epsilon) \to \mathbf{0} \quad \texttt{size}(\mathbf{i}(x)) \to \mathbf{S}(\texttt{size}(x))$$

□

Remarque :
La quasi-interprétation d'un polynôme se confond avec sa sémantique. C'est à mon avis une propriété intrinsèque des quasi-interprétations, liée à la taille des termes calculés. Ce point de vue est développé un peu plus en détail ci-dessous.

Théorème 39.
Une fonction calculable en temps polynômial par une PRM est calculable par un système de réécriture qui termine par LPO et admet une quasi-interprétation.

Démonstration.
Ceci découle des lemmes 37 et 38. □

Toute fonction de Pspace est donc calculable par une PRM en temps polynômial, donc par un système de réécriture qui termine par LPO et admet une quasi-interprétation.

5.3 Quasi-interprétations et taille

Comme on l'a vu, pour les polynômes la quasi-interprétation et la sémantique se confondent. En fait, on peut aussi remarquer que si t est un terme constructeurs représentant l'entier n en unaire (c'est-à-dire à partir de **0** et **S**) et P un polynôme (ou le système de réécriture le calculant), on a $n = |t| = (\!|t|\!)$ et $P(n) = |P(t)| = (\!|P(t)|\!)$. Je pense que ce n'est absolument pas une coïncidence et que pour toute fonction de $\mathbb{N}^m$ vers $\mathbb{N}$ la sémantique fournit une quasi-interprétation.

En fait, cette propriété peut, à mon avis, se généraliser à n'importe quel système en considérant la taille des termes.

Affirmation 40.
Soit main un système de réécriture. Si pour tous les termes constructeurs $t, t_1, \ldots, t_n$ tels que t soit la forme normale de `main`$(t_1, \ldots, t_n)$ *on a $|t| = f(|t_1|, \ldots, |t_n|)$ alors il existe un système de réécriture qui calcule la même fonction et qui admet f comme quasi-interprétation.*

Ce fait prend tout son intérêt si on le compare aux travaux récents de M. Hofmann, H. Schwichtenberg et K. Aehlig [Hof99, AS00, Hof02] sur les fonctions "Non-Size Increasing". En effet, ils considèrent les fonctions dont la taille du résultat est bornée par la taille des entrées et montrent qu'elles ont plusieurs propriétés intéressantes et que l'incrément de taille est souvent responsable des temps de calculs exponentiels. En particulier, ces fonctions ont la particularité d'être calculables *en place*, c'est-à-dire sans avoir besoin d'allouer de la mémoire au cours de l'exécution. On peut donc, par exemple, les compiler vers un programme `C` qui ne contiendrait aucun `malloc` et se contenterait de réutiliser les pointeurs et la mémoire dont il dispose au début. Ce qui est extrêmement intéressant dans l'optique des "codes mobiles" envoyés sur internet et exécutés sur des machines distantes : on est sûr que le code reçu ne va pas accéder à des zones mémoires non autorisées.

Mais revenons aux quasi-interprétations. Si une fonction est Non-Size Increasing, la taille du résultat est donc bornée par la taille des entrées. Elle est donc calculable par un système de réécriture qui admet une quasi-interprétation bornée par la taille des entrées, c'est-à-dire à peu de chose près une quasi-interprétation dans l'algèbre $(\max, +)$. Or c'est justement cette classe de quasi-interprétations qui a été étudiée par R. Amadio [Ama02] et qui permet donc de donner une autre caractérisation des fonctions Non-Size Increasing en terme de quasi-interprétations et plus en terme de systèmes de types comme le fait M. Hofmann.

Un autre intérêt de ce fait est qu'il donne peut-être une piste pour calculer les quasi-interprétations. En effet, il est indécidable de savoir si un système de réécriture admet une interprétation polynomiale mais les quasi-interprétations sont peut-être suffisamment différentes des interprétations pour être décidables (on peut notamment penser à `lcs` qui n'admet aucune interprétation mais admet une quasi-interprétation).

Étudier les modifications subies par la taille des termes est une approche possible pour étudier la décidabilité des quasi-interprétations. Et cette étude est déjà relativement avancée. C'est en effet sur ce critère que C. S. Lee, N. D. Jones et A. M. Ben-Amram ont basé leur "Size-Change Principle" [LJBA01] et que, avec J.-Y. Marion, nous avons développé la "Terminaison par ressources" en utilisant des réseaux de Petri (voir les chapitres suivants à ce sujet).

Chapitre 6

Réseaux de Petri

Les réseaux de Petri ont été introduits par C. A. Petri vers 1970. Ils sont assez couramment utilisés pour modéliser des systèmes dynamiques qui doivent partager des ressources, comme par exemple deux sémaphores (exclusion mutuelle). Il est donc relativement naturel de penser à eux pour représenter les ressources dont dispose un programme : temps ou espace. En l'occurrence, je les utilise pour modéliser l'utilisation de l'espace par une machine à compteurs et l'étude du réseau de Petri va permettre de détecter deux propriétés : la terminaison et le "calcul en place".

Ce chapitre présente rapidement les notions de base sur les réseaux de Petri. Les notations sont pour la plupart empruntées au livre *Free-choice Petri nets* de J. Desel et J. Esparza ([DE95]).

6.1 Réseaux

Définition 82 (Réseaux).
Un Réseau de Petri N est un triplet (S, T, F) où S et T sont deux ensembles finis disjoints et F est une relation binaire sur $S \bigcup T$ telle que $F \bigcap (S \times S) = F \bigcap (T \times T) = \emptyset$.

Les éléments de S sont appelés *places* et seront graphiquement représentés par des cercles ou des ellipses. Les éléments de T sont appelés *transitions* et seront graphiquement représentés par des rectangles. F est la *relation de flot* du réseau et sera graphiquement représentée par des flèches entre les places et les transitions ou entre les transitions et les places. Les éléments de $S \bigcup T$, quand il n'est pas important de savoir si c'est une place ou une transition, sont appelés des *nœuds*.

D'un point de vue de théorie des graphes, $((S \bigcup T), F)$ est un graphe bipartite orienté.

Pour plus de commodité, les noms des places et des transitions seront écrits à l'intérieur d'icelles dans les représentations graphiques.

Exemple 15.
La figure 6.1 représente un réseau de Petri dont les places sont l'ensemble $S = \{p_1, p_2, p_3, p_4, p_5\}$, les transitions sont l'ensemble $T = \{t_1, t_2, t_3, t_4, t_5\}$ et la relation de flot est
$F = \{(p_1, t_2), (t_2, p_2), (p_2, t_1), (t_1, p_1), (t_2, p_3), (p_3, t_4), (t_4, p_5), (p_5, t_3), (t_3, p_3), (p_5, t_5), (t_5, p_4), (p_4, t_2)\}$.

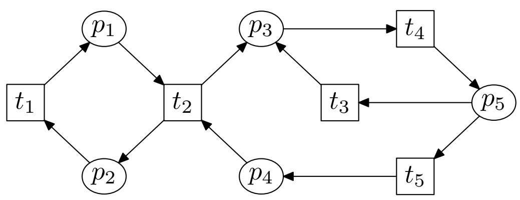

FIG. 6.1 – Un réseau de Petri.

Définition 83 (Préset, postset).
Soient $N = (S, T, F)$ un réseau de Petri et a un nœud de N. L'ensemble ${}^\bullet a = \{b/(b,a) \in F\}$ est le *préset* de a. L'ensemble $a^\bullet = \{b/(a,b) \in F\}$ est le *postset* de a.

Les transitions dans le préset (resp. le postset) d'une place sont appelées ses transitions *d'entrée* (resp. *de sortie*). Les places dans le préset (resp. le postset) d'une transition sont appelées ses places *d'entrée* (resp. *de sortie*).

Soit A un ensemble de nœuds. Les pré- et postsets sont étendus aux ensembles de nœuds de façon canonique : ${}^\bullet A = \bigcup_{a \in A} {}^\bullet a$ et $A^\bullet = \bigcup_{a \in A} a^\bullet$.

Remarque :
D'une manière plus générique, on peut s'intéresser au cas où les pré- et postsets sont non plus des ensembles mais des multisets (*ie* chaque nœud peut apparaître plusieurs fois dans le pré- ou postset d'un autre nœud). Ce qui peut aussi être facilement représenté en assignant des poids (correspondants à la multiplicité) à chaque arc de la relation de flot.

Exemple 16.
Dans le réseau dessiné précédemment on a, entre autre, ${}^\bullet p_1 = \{t_1\}$, $p_5^\bullet = \{t_3, t_5\}$, $t_2^\bullet = \{p_2, p_3\}$, ${}^\bullet\{t_1, p_3\} = \{p_2, t_2, t_3\}$.

Définition 84 (Chemins).
Soit $N = (S, T, F)$ un réseau de Petri. Un *chemin* dans N est une suite finie non vide de nœuds $a_1, \ldots, a_{k+1}$ telle que pour tout $1 \leq i \leq k$, $(a_i, a_{i+1}) \in F$.

Par la suite, nous nous intéresserons uniquement aux réseaux connexes, c'est-à-dire ceux pour lesquels le graphe bipartite sous-jacent est connexe.

6.2 Marquages, tirs, exécutions

Un marquage met des jetons dans les places.

Définition 85 (Marquages).
Soit $N = (S, T, F)$ un réseau de Petri. Un *marquage* de N est une fonction M des places vers les entiers positifs.

Soit M un marquage de N. La place s est marquée à M si $M(s) > 0$. Le marquage vide est celui qui assigne 0 à chaque place, il est noté $\mathbf{0}$.

Soient N un réseau, M un marquage et s une place. On dit que la place s contient $M(s)$ *jetons*.

Graphiquement, un marquage est représenté en mettant dans chaque place autant de points que le nombre de jetons contenus.

Le nombre total de jetons contenus dans un ensemble de places est la somme des nombres de jetons contenus dans chaque place : $M(R) = \sum_{s \in R} M(s)$.

Les marquages sont partiellement ordonnés en fonction du nombre de jetons dans chaque place. c'est-à-dire que $M \trianglelefteq M'$ si et seulement si pour chaque place s, $M(s) \leq M'(s)$.

Un marquage active une transition si il met au moins un jeton dans chaque place en amont. on peut alors tirer la transition, ce qui enlève un jeton de chaque place en amont et ajoute un jeton dans chaque place en aval.

Définition 86 (Transitions actives).
Soient N un réseau, M un marquage et t une transition. On dit que M *active* t (ou que t est activée à M) si M marque toutes les places dans le préset de t, c'est-à-dire $\forall s \in^\bullet t, M(s) > 0$.

Si t est active au marquage M, elle peut être *tirée*. Le tir de t amène au marquage successeur M' qui est défini à partir de M en enlevant un jeton de chaque place dans le préset de t et en en ajoutant un dans chaque place du postset de t, c'est-à-dire :

$$M'(s) = \begin{cases} M(s) - 1 & \text{si } s \in^\bullet t \text{ et } s \notin t^\bullet \\ M(s) + 1 & \text{si } s \in t^\bullet \text{ et } s \notin^\bullet t \\ M(s) & \text{sinon} \end{cases}$$

On note $M\xrightarrow{t}M'$.

Remarque :
Un marquage peut activer plusieurs transitions. Dans ce cas, le choix de la transition à tirer est fait de manière non déterministe.

Le nombre de jetons présents dans le réseau peut varier si il existe des transitions qui n'ont pas le même nombre de places d'entrée et de places de sortie.

Définition 87 (Séquence de tirs, atteignabilité).
Soit N un réseau. Si il existe une suite (finie ou infinie) de transitions t_i et de marquages M_i telles que pour tout i, M_i active t_i et $M_i\xrightarrow{t_i}M_{i+1}$ alors on dit que $\omega = t_0, \dots$ est une ***séquence de tirs*** (finie ou infinie).

Si ω est finie, on dit qu'elle mène de M_0 à M_n et on note $M_0\xrightarrow{\omega}M_n$. Si ω est infinie, on note $M_0\xrightarrow{\omega}$. ϵ est la séquence de tirs vide, et pour tout marquage M, on a $M\xrightarrow{\epsilon}M$.

ω est *active* à M si ω est finie et il existe un marquage M' tel que $M\xrightarrow{\omega}M'$; ou ω est infinie et $M\xrightarrow{\omega}$.

M' est *atteignable* à partir de M si il existe une séquence de tirs ω telle que $M\xrightarrow{\omega}M'$. On note alors $M\xrightarrow{*}M'$.

6.3 Terminaison

Définition 88 (Impasses, exécutions).
Soient N un réseau et M un marquage. M est une *impasse* si il n'active aucune transition.
Soient N un réseau et M un marquage. Une *exécution* ω partant de M est une séquence de tirs maximale partant de M. C'est-à-dire que soit ω est infinie et $M\xrightarrow{\omega}$, soit ω est finie, $M\xrightarrow{\omega}M'$ et M' est une impasse.

Définition 89 (Terminaison).
Un réseau de Petri *termine* si il n'existe aucune exécution infinie (quelque soit le marquage initial). Si il existe un marquage qui active une exécution infinie, le réseau est *non-terminant*.

Un réseau de Petri est *fortement non-terminant* si il existe un marquage qui active une exécution infinie dans laquelle chaque transition apparaît une infinité de fois.

Définition 90 (Vivacité).
Soient N un réseau et M_0 un marquage. N est *vivant* à M_0 si chaque transition pourra toujours être tirée. C'est-à-dire que pour chaque marquage M_i atteignable depuis M_0 et pour chaque transition t, il existe une séquence de tirs ω telle que $M_i\xrightarrow{\omega}M'$ et M' active t.

D'une manière plus générale, on dit que le réseau est vivant si il existe un marquage M tel que le réseau soit vivant à M.

Lemme 41.
La vivacité est plus forte que l'absence d'impasses.

Démonstration.
Le réseau de la figure 6.2 ne comporte pas d'impasse mais il n'est pas vivant. Le réseau est sans impasse grâce à la transition t_2 qui boucle. Mais il n'est pas vivant car si il n'y a plus de jetons dans p_1, t_1 ne pourra plus être tirée. □

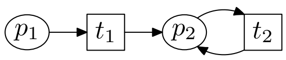

FIG. 6.2 – Un réseau non-vivant et sans impasses.

Lemme 42.
Un réseau de Petri vivant est fortement non-terminant.

Démonstration.
Soient M_0 un marquage tel que N soit vivant à M_0 et $t_1, \ldots, t_n$ une énumération des transitions. Par définition de la vivacité, il existe une séquence de tirs ω_1 et un marquage M_1 tels que $M_0 \xrightarrow{\omega_1} M_1$ et M_1 active t_1. On construit la séquence $\omega'_1 = \omega_1 t_1$ qui conduit de M_0 à M'_1.

Par définition de la vivacité, il existe une séquence ω_2 et un marquage M_2 tels que $M'_1 \xrightarrow{\omega_2} M_2$ et M_2 active t'. On construit alors la séquence $\omega'_2 = \omega'_1 \omega_2 t_2$ qui conduit de M_0 à M'_2.

On continue de la sorte en construisant une séquence qui ira tirer t_3, puis $\ldots$, puis t_n, puis de nouveau $t_1, \ldots$ L'exécution ainsi obtenue comporte chaque transition une infinité de fois. □

Lemme 43.
La réciproque n'est pas vraie.

Démonstration.
Le réseau de Petri dessiné à la figure 6.3 n'est pas vivant. En effet, si à chaque fois qu'un jeton est en p_1 on tire la transition t_2, on fini par obtenir un marquage où tous les jetons sont en p_3 qui est une impasse.

Cependant, le réseau est fortement non-terminant. En effet, si on tire la séquence $t_1 t_2 t_3$, on se retrouve dans le même marquage. on peut donc tirer cette séquence en boucle, obtenant ainsi une séquence infinie qui tire chacune des trois transitions une infinité de fois. □

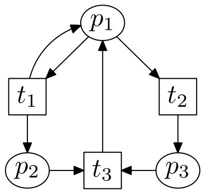

FIG. 6.3 – Un réseau de Petri fortement non terminant mais non-vivant.

6.4 L'aspect algébrique

Afin de manipuler plus aisément les réseaux de Petri et leurs propriétés, nous introduisons ici un formalisme algébrique qui permet de ramener beaucoup de problèmes à des calculs matriciels.

La plupart des nouvelles notations de cette partie sont reprises de l'article "Termination properties of generalized Petri nets" de Y. E. Lien ([Lie76]).

Définition 91 (Vecteurs et matrices caractértistiques).
Soit $N = (S, T, F)$ un réseau de Petri. On se donne une énumération des places et une énumération des transitions.

Soit M un marquage. On le représente par un vecteur de $\mathbb{N}^{|S|}$ dont chaque composante est le nombre de jetons dans la place correspondante (en accord avec l'énumération choisie). On notera aussi M le vecteur, le contexte permettant de décider si il s'agit d'un marquage ou du vecteur le représentant.

Soit t une transition. On la représente par un vecteur de $\mathbb{Z}^{|S|}$ dont chaque composante est le nombre de jetons gagnés (si positive) ou perdus (si négative) par la place correspondante (en accord avec l'énumération des places) lorsqu'on tire la transition. De la même manière, on notera aussi t le vecteur représentant la transition.

La *matrice caractéristique* de N est la matrice de $\mathbb{Z}^{|S| \times |T|}$ dont chaque colonne est un vecteur représentant une transition (dans l'ordre de l'énumération choisie). On la nomme Γ_N ou tout simplement Γ si le contexte est clair.

Soit ω une séquence de tirs, son *vecteur caractéristique* (ou vecteur de Parikh) est le vecteur de $\mathbb{N}^{|T|}$ dont chaque composante indique le nombre d'occurrence dans ω de la transition correspondante (en accord avec l'énumération choisie). On le note F_ω.

Exemple 17.
La matrice caractéristique du réseau de Pétri de la figure 6.1 est la suivante :

	t_1	t_2	t_3	t_4	t_5
p_1	1	-1	0	0	0
p_2	-1	1	0	0	0
p_3	0	1	1	-1	0
p_4	0	-1	0	0	1
p_5	0	0	-1	1	-1

Lemme 44.
Soient N un réseau de Petri, M, M' deux marquages et t une transition telle que $M \xrightarrow{t} M'$. On a l'équation matricielle $M + t = M'$.

Démonstration.
Évident par construction des vecteurs. □

Lemme 45.
Soient N un réseau, M, M' deux marquages et ω une séquence de tirs telle que $M \xrightarrow{\omega} M'$. On a l'équation matricielle $M + \Gamma_N \times F_\omega = M'$.

Démonstration.
Par récurrence sur la longueur de ω.
- Si $\omega = t$ ne comporte qu'une transition, on a $\Gamma_N \times F_\omega = t$ et l'équation est obtenue par le lemme précédent.
- Si $\omega = \omega' t$ alors par définition de la séquence de tirs, il existe un marquage A tel que $M \xrightarrow{\omega'} A$ et $A \xrightarrow{t} M'$. Par construction du vecteur caractéristique, on a $F_\omega = F_{\omega'} + F_t$. Par hypothèse de récurrence, on a $M + \Gamma_N \times F_{\omega'} = A$ et $A + \Gamma_N \times F_t = M'$. Donc $M + \Gamma_N \times F_\omega = M'$.

□

Chapitre 7

Terminaison par ressources

La terminaison par ressources étudie de manière précise la gestion de l'espace au cours de l'exécution d'un programme et en particulier l'évolution de la taille des variables. L'idée est reprise du "Size-Change Principle" de C. S. Lee, N. D. Jones et A. M. Ben-Amram [LJBA01] dont je rappelle les grandes idées ici.

7.1 Size-Change Principle

Le Size-Change Principle (SCP) étudie la terminaison des systèmes de réécriture en partant du principe général suivant : si la réduction d'un terme est infinie, il se produit une infinité d'appels de fonctions dans cette réduction. Ainsi, en étudiant toutes les chaînes d'appels de fonctions plausibles (certaines ne peuvent avoir lieu en réalité), et en montrant qu'aucune d'elles ne peut être infinie, on peut prouver la terminaison du système de réécriture.

Pour ça, il faut commencer par identifier les "call sites", c'est-à-dire tous les endroits où une fonction en appelle une autre, et les numéroter. Ensuite, pour chacun de ces appels, on construit un "Size-Change Graph" qui indique comment telle variable de la fonction appelante devient telle variable de la fonction appelée. Il y a trois cas possibles.

- Soit une variable est passée telle quelle à la fonction appelée (comme par exemple dans la règle $\mathtt{f}(x) \to \mathtt{g}(x)$). Dans ce cas, on place une flèche entre les variables et on obtient la brique suivante : $\boxed{\mathtt{f}_1 \longrightarrow \mathtt{g}_1}$ qui signifie que la première variable de $\mathtt{g}$ est égale à la première variable de $\mathtt{f}$.
- Soit une variable est passée avec une décroissance à la fonction appelée (comme par exemple dans la règle $\mathtt{f}(x+1) \to \mathtt{g}(x)$). Dans ce cas, on indique cette décroissance sur la flèche reliant les variables, et on obtient la brique $\boxed{\mathtt{f}_1 \xrightarrow{\downarrow} \mathtt{g}_1}$ indiquant que la première variable de $\mathtt{g}$ est strictement plus petite que la première variable de $\mathtt{f}$.
- Dans les autres cas (variable qui croît comme dans $\mathtt{f}(x) \to \mathtt{g}(x+1)$ ou variable qui dépend de plusieurs autres ou du résultat d'un calcul comme dans $\mathtt{f}(x,y) \to \mathtt{g}(\mathtt{h}(x,y))$, on ne met aucune flèche arrivant sur la variable de $\mathtt{g}$ car on ne sait rien sur sa valeur. On obtient ainsi la brique $\boxed{\mathtt{f}_1 \quad \mathtt{g}_1}$ (dans le deuxième cas, il y aurait des flèches d'égalité dans la brique décrivant l'appel de $\mathtt{h}$ par $\mathtt{f}$).

Considérons par exemple le système de réécriture suivant (les deux "call sites" ont été identifiés et numérotés) :

$$\begin{array}{rcl} \mathtt{f}(x,\mathbf{0}) & \to & x \\ \mathtt{f}(\mathbf{0},\mathbf{S}(y)) & \to & 1 : \mathtt{f}(\mathbf{S}(y),y) \\ \mathtt{f}(\mathbf{S}(x),\mathbf{S}(y)) & \to & 2 : \mathtt{f}(\mathbf{S}(y),x) \end{array}$$

On peut remarquer que ce système ne termine ni par MPO, ni par LPO. Les deux Size-Change Graphs obtenus pour ce système sont dessinés sur la figure 7.1.

Ensuite, il faut construire toutes les chaînes d'appels infinies potentielles. Pour ça, on regarde uniquement les fonctions appelées et appelantes, sans s'occuper des valeurs des arguments. Ainsi, dans l'exemple ci-dessus, le mot 11 représente une chaîne d'appels "légale" bien qu'elle ne puisse avoir lieu en réalité (car après un appel 1, la première variable ne peut pas valoir $\mathbf{0}$ donc la deuxième règle de réécriture ne peut

Fig. 7.1 – Les deux Size-Change Graphs du système étudié.

pas être appliquée de nouveau et on ne peut pas avoir un nouvel appel 1). Cependant, considérer toutes les chaînes d'appels en se concentrant uniquement sur les fonctions et non sur les valeurs des variables permet de les construire facilement.

Dans l'exemple ci-dessus, ces chaînes infinies sont formées des appels 1 ou 2 dans n'importe quel ordre (puisqu'il n'y a qu'une seule fonction). Elles peuvent donc être représentées par l'expression $\{1+2\}^\omega$.

Parmi ces chaînes d'appels, on peut partir d'une variable et suivre le "fil" constitué par les flèches placées dans les Size-Change Graphs. Ce fil peut être fini ou infini. Si il existe un tel fil infini qui, de plus, comporte une infinité de décroissances (*ie* de $\xrightarrow{\downarrow}$) alors la chaîne d'appels ne pourrait pas être exécutée par le programme. En effet, son exécution signifierait une infinité de décroissances sur la valeur d'une variable, ce qui est impossible (l'ordre étant bien fondé).

Ainsi, si dans chaque chaîne d'appels infinie il existe un fil comportant une infinité de décroissance, on peut en déduire que le programme termine toujours.

Pour l'exemple précédent, chaque chaîne finie de la forme 12^* (c'est à dire un appel 1 et plusieurs appels 2) contient un fil qui va de $\texttt{f}_2$ à $\texttt{f}_2$ avec au moins une décroissance. On peut donc considérer deux cas :

- Soit la chaîne contient une infinité d'appels 1 auquel cas il existe un fil avec une infinité de $\xrightarrow{\downarrow}$ sur $\texttt{f}_2$.
- Sinon, il y a un moment à partir duquel la chaîne ne contient plus que des appels 2 et à ce moment là on a deux fils qui se croisent tout le temps et qui comportent tous les deux une infinité de $\xrightarrow{\downarrow}$.

Donc chaque chaîne d'appels infinie contient un fil avec une infinité de décroissances, et ne peut donc pas correspondre à une vraie chaîne de réduction infinie du système de réécriture. Donc le système termine toujours.

Il est assez facile de construire un automate de Büchi (automate sur les mots infinis) qui reconnaît les mots décrivant une chaîne d'appels légale. Il est aussi possible de construire un automate de Büchi qui reconnaît les mots décrivant une chaîne avec un fil contenant une infinité de $\xrightarrow{\downarrow}$ (cette construction est un peu plus délicate mais pas très difficile). Tester si deux automates de Büchi reconnaissent le même langage se fait en espace polynomial.

C. S. Lee et al. ont aussi prouvé que leur méthode est effectivement Pspace-complète.

Le principal inconvénient de cette méthode est de ne suivre que les décroissances des valeurs et d'abandonner tout espoir de terminaison à la moindre croissance. Ainsi le système de réécriture suivant suffit à faire échouer le Size-Change Principle :

$$
\begin{array}{rcl}
\texttt{f}(x) & \rightarrow & 1 : \texttt{g}(\mathbf{S}(x)) \\
\texttt{g}(\mathbf{0}) & \rightarrow & \mathbf{0} \\
\texttt{g}(\mathbf{S}(\mathbf{0})) & \rightarrow & \mathbf{0} \\
\texttt{g}(\mathbf{S}(\mathbf{S}(x))) & \rightarrow & 2 : \texttt{f}(x)
\end{array}
$$

En effet, on aurait deux Size-Change Graphs :

- $\boxed{\texttt{f}_1 \quad \texttt{g}_1}$ pour le premier appel.
- $\boxed{\texttt{g}_1 \xrightarrow{\downarrow} \texttt{f}_1}$ pour le deuxième.

Les chaînes infinies sont composées d'un appel 1 et d'un appel 2 alternativement. Or ces chaînes ne comportent aucun fil infini car l'appel 1 "coupe" le fil à cause de la croissance.

Pourtant, le système termine toujours car la valeur de x augmente de 1 puis diminue de 2, avec pour effet net une diminution de 1.

En outre, cette méthode gère mal les appels de fonctions imbriqués. Ainsi, pour prouver la terminaison du tri rapide (et, par extension, de la plupart des algorithmes "Diviser pour régner") il faut une preuve annexe sur les propriétés de `split` (qui revient principalement à dire que les deux projections font décroître la taille de leurs arguments ce qui ramène de façon intéressante le problème des fonctions Non-Size Increasing). Une comparaison entre entiers a été ajoutée au langage pour plus de commodité, avec la sémantique attendue. Il est assez facile de voir qu'on peut programmer cet algorithme sans cette extension (mais au prix d'un système de réécriture plus compliqué).

$$\begin{array}{rcl}
\texttt{sort}(\mathbf{Nil}) & \to & \mathbf{Nil} \\
\texttt{sort}(\mathbf{Cons}(x,\mathbf{l})) & \to & \texttt{sortpair}(x,\texttt{split}(\mathbf{l})) \\
\texttt{sortpair}(x,\mathbf{Pair}(\mathbf{l},\mathbf{l}')) & \to & \texttt{concat}(\texttt{sort}(\mathbf{l}),x,\texttt{sort}(\mathbf{l}'))
\end{array}$$

$$\begin{array}{rcl}
\texttt{split}(x,\mathbf{Nil}) & \to & \mathbf{Pair}(\mathbf{Nil},\mathbf{Nil}) \\
\texttt{split}(x,\mathbf{Cons}(y,\mathbf{l})) & \to & \texttt{if } x > y \\
 & & \quad \texttt{then cons1}(y,\texttt{split}(x,\mathbf{l})) \\
 & & \quad \texttt{else cons2}(y,\texttt{split}(x,\mathbf{l}))
\end{array}$$

$$\begin{array}{rcl}
\texttt{cons1}(x,\mathbf{Pair}(\mathbf{l},\mathbf{l}')) & \to & \mathbf{Pair}(\mathbf{Cons}(x,\mathbf{l}),\mathbf{l}') \\
\texttt{cons2}(x,\mathbf{Pair}(\mathbf{l},\mathbf{l}')) & \to & \mathbf{Pair}(\mathbf{l},\mathbf{Cons}(x,\mathbf{l}'))
\end{array}$$

$$\begin{array}{rcl}
\texttt{concat}(\mathbf{Nil},y,\mathbf{l}') & \to & \mathbf{Cons}(y,\mathbf{l}') \\
\texttt{concat}(\mathbf{Cons}(x,\mathbf{l}),y,\mathbf{l}') & \to & \mathbf{Cons}(x,\texttt{concat}(\mathbf{l},y,\mathbf{l}'))
\end{array}$$

De même, un exemple comme l'algorithme d'Euclide [Euc00], dont la terminaison repose lourdement sur les propriétés de la soustraction, a besoin d'une preuve annexe pour que la terminaison soit détectée par le Size-Change Principle.

$$\begin{array}{rcl}
\texttt{pgcd}(x,\mathbf{0}) & \to & x \\
\texttt{pgcd}(x,y) & \to & \texttt{if } x < y \\
 & & \quad \texttt{then pgcd}(y-x,x) \\
 & & \quad \texttt{else pgcd}(x-y,y)
\end{array}$$

$$\begin{array}{rcl}
x - \mathbf{0} & \to & x \\
\mathbf{S}(x) - \mathbf{S}(y) & \to & x - y
\end{array}$$

7.2 Size-Change Petri Net

Le problème du Size-Change Principle est de regarder ce que deviennent les variables de façon très locale. La terminaison par ressources essaie de regarder les choses de façon plus globale et de suivre un peu plus ce qui se passe. Pour ce faire, on simule le comportement des machines à compteurs par des réseaux de Petri. Chaque variable est représentée par une place, la taille de la variable devient alors le nombre de jetons dans la place et le fonctionnement des réseaux de Petri permet de suivre sans difficultés ce qui se passe. Une telle simulation a déjà été faite, mais j'ajoute ici des places qui permettent de prendre en compte les ressources, idée assez logique dans le cadre des travaux sur la complexité implicite des calculs et qu'on peut aussi aussi retrouver, par exemple, dans la terminaison par induction de O. Fissore, I. Gnaedig et H. Kirchner [FGK02], les paires de dépendance de T. Arts et J. Giesl [AG00] ou la supercompilation (voir par exemple la thèse de J. P. Secher qui résume bien la problématique et les techniques employées [Sec02]).

Le choix des machines à compteurs plutôt que des systèmes de réécriture comme objet d'étude est dû au fait que leur comportement élémentaire (l'exécution d'une instruction) est beaucoup plus simple que celui d'un système de réécriture (un pas de réduction) et ils se prêtent donc beaucoup mieux à une modélisation par réseaux de Petri. De plus, cette modélisation a déjà été utilisée, par exemple par J. Esparza [Esp98]. Elle est aussi le prolongement naturel d'une généralisation dont Neil D. Jones m'a parlé lors de mon séjour à Copenhague à l'été 2002 pour étendre le Size-Change Principle aux fonctions à plusieurs résultats qui "*seems to make the Size-Change analysis easier to understand*".

Je présente ici les idées directrices de cette technique d'analyse. Les preuves complètes des théorèmes, ainsi que la description précise des systèmes d'inéquations à résoudre, sont assez délicates. J'ai préféré les regrouper au dernier chapitre pour éviter de noyer l'intuition sous ces aspects techniques.

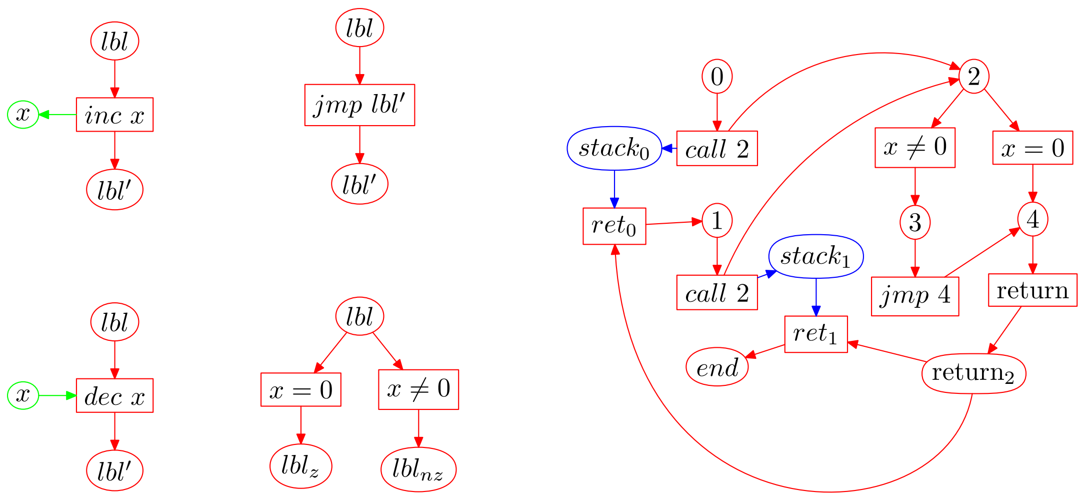

FIG. 7.2 – SCPN des instructions.

7.2.1 Des machines aux réseaux

Le “Size-Change Petri Net” (SCPN) d’une machine permet de simuler les exécutions de cette machine de manière non-déterministe. L’idée est que chaque transition du SCPN tirée correspond à l’exécution d’une instruction de la machine à compteurs. Le SCPN contient une place pour chaque label, une place pour chaque variable et quelques places supplémentaires pour gérer la pile de retour (en gros, une place pour chaque sous-routine et une pour chaque `call`). Il y a une transition pour chaque instruction différente de `jz` et deux transitions pour chaque `jz`.

Lors de la simulation de l’exécution d’une machine, il n’y a en permanence qu’un seul jeton dans les places correspondant aux labels. La place qui contient ce jeton correspondant au label où se trouve le contrôle de l’exécution. Ce jeton est donc assimilable au compteur ordinal (“Instruction Pointer”) des ordinateurs.

Quand il y a des sous-routines, le SCPN envoie un jeton dans la sous-routine et en conserve un dans une place qui sert de pile, pour savoir où doit avoir lieu le retour. Quand la sous-routine est finie, la transition située après cette place est activée et le programme ne peut donc reprendre que à l’endroit où il était avant l’appel à la sous-routine.

La figure 7.2 montre les SCPN des différentes instructions et sert de définition informelle du SCPN d’un réseau complet. La partie rouge correspond au “Control Flow”, la partie verte correspond aux variables et la partie bleue à la gestion de la mémoire (pour l’instant limitée à la pile de retour). Afin de ne pas mélanger les adresses de retour de différents `call`, on a en fait une pile pour chaque `call`. Le réseau de Petri correspondant au `call` est construit d’après le programme suivant :

$$\begin{array}{rcl} 0 & : & \texttt{call}\ 2 \\ 1 & : & \texttt{call}\ 2 \\ \multicolumn{3}{l}{\mathit{end}} \\ 2 & : & \texttt{jz}\ x\ 4 \\ 3 & : & \texttt{jmp}\ 4 \\ 4 & : & \texttt{return} \end{array}$$

Par la suite, si une machine comporte plusieurs fois la même instruction (comme par exemple le `call` 2 pour la machine ci-dessus), ces instructions seront implicitement différenciées par les labels auxquels elles sont situées. Les places ou transitions des réseaux porteront le même nom que les labels, variables ou instructions correspondants dans la machine à compteurs.

Définition 92.
Soit `pgm` une machine à compteurs. On définit une fonction `next` des labels vers les labels qui renvoie le label immédiatement après dans le programme. C’est-à-dire que si la machine comporte les labels et les instructions $\texttt{lbl}_0 : \texttt{inst}_0; \ldots; \texttt{lbl}_n : \texttt{inst}_n$ alors pour tout $i < n$ on a $\texttt{next}(\texttt{lbl}_i) = \texttt{lbl}_{i+1}$.

Définition 93 (SCPN).
Soit pgm une machine à compteurs. Le *SCPN* de pgm est un réseau de Petri $N = (S, T, F)$ tel que :
- Il existe une place lbl pour chaque label, une place x pour chaque variable, une place $\texttt{return}_{\texttt{lbl}}$ pour chaque sous-routine où lbl est le premier label de la sous-routine (celui vers lequel sont dirigés les calls) et une place $stack_{\texttt{lbl}}$ pour chaque call (lbl est le label où est situé le call).
- Il existe une transition inst pour chaque instruction différente de jz, deux transitions $x = 0$ et $x \neq 0$ pour chaque instruction jz x lbl et une transition $ret_{\texttt{lbl}}$ pour chaque call (lbl est le label où est situé le call).
- Si $\texttt{instr(lbl)} = \texttt{inc}\ x$ alors
  - $(\texttt{inc}\ x) \in \texttt{lbl}^\bullet$.
  - $(\texttt{inc}\ x) \in^\bullet x$.
  - $(\texttt{inc}\ x) \in^\bullet \texttt{next(lbl)}$.
- Si $\texttt{instr(lbl)} = \texttt{dec}\ x$ alors
  - $(\texttt{dec}\ x) \in \texttt{lbl}^\bullet$.
  - $(\texttt{dec}\ x) \in x^\bullet$.
  - $(\texttt{dec}\ x) \in^\bullet \texttt{next(lbl)}$.
- Si $\texttt{instr(lbl)} = \texttt{jmp lbl}'$ alors
  - $(\texttt{jmp lbl}') \in \texttt{lbl}^\bullet$.
  - $(\texttt{jmp lbl}') \in^\bullet \texttt{lbl}'$.
- Si $\texttt{instr(lbl)} = \texttt{jz}\ x\ \texttt{lbl}'$ alors
  - $(x = 0), (x \neq 0) \in \texttt{lbl}^\bullet$.
  - $(x \neq 0) \in^\bullet \texttt{next(lbl)}$
  - $(x = 0) \in^\bullet \texttt{lbl}'$
- Si $\texttt{instr(lbl)} = \texttt{return}$ alors
  - $(\texttt{return}) \in \texttt{lbl}^\bullet$.
  - $(\texttt{return}) \in^\bullet \texttt{return}_{\texttt{lbl}'}$ où $\texttt{lbl}'$ est le premier label de la sous-routine qui contient ce return.
- Si $\texttt{instr(lbl)} = \texttt{call lbl}'$ alors
  - $(\texttt{call lbl}') \in \texttt{lbl}^\bullet$.
  - $(\texttt{call lbl}') \in^\bullet \texttt{lbl}'$.
  - $(\texttt{call lbl}') \in^\bullet stack_{\texttt{lbl}}$.
  - $(ret_{\texttt{lbl}}) \in^\bullet \texttt{next(lbl)}$
  - $(ret_{\texttt{lbl}}) \in stack(\texttt{lbl})^\bullet$
  - $(ret_{\texttt{lbl}}) \in \texttt{return}^\bullet_{\texttt{lbl}'}$

Définition 94 (Places ordinales).
Soient pgm une machine à compteurs et N son SCPN. Les *places ordinales* de N sont les places $\texttt{return}_{\texttt{lbl}}$ et celles qui correspondent aux labels de pgm.

Le *CFPN (Control Flow Petri Net)* est le réseau de Petri composé des places ordinales et des transitions du SCPN.

Tout jeton dans une place ordinale est appelé *jeton ordinal*. Si il n'y a qu'un seul jeton dans l'ensemble de toutes les places ordinales, ce jeton est alors appelé *compteur ordinal*.

Le qualificatif "ordinal" vient du compteur ordinal des langages assembleur, qui indique quelle instruction doit être exécutée. En effet, les places ordinales correspondent exactement aux endroits du programme où peut être positionné le compteur ordinal. Dans tous les schémas, le CFPN est représenté en rouge.

Lemme 46 (S-invariant).
Le nombre de jetons dans les places ordinales est constant lors d'une exécution.

Démonstration.
Chaque transition a exactement une place ordinale en entrée et une en sortie. □

Cette propriété est ce qu'on appelle un S-invariant du réseau de Petri. L'un des intérêts de ces invariants est d'obtenir une condition nécessaire à l'accessibilité d'un marquage depuis un autre.

Corollaire 47 (Condition nécessaire d'accessibilité).
Soient pgm *une machine à compteurs, N son SCPN, M et M' deux marquages de N. Si M' est accessible depuis M alors M et M' mettent autant de jetons dans les places ordinales.*

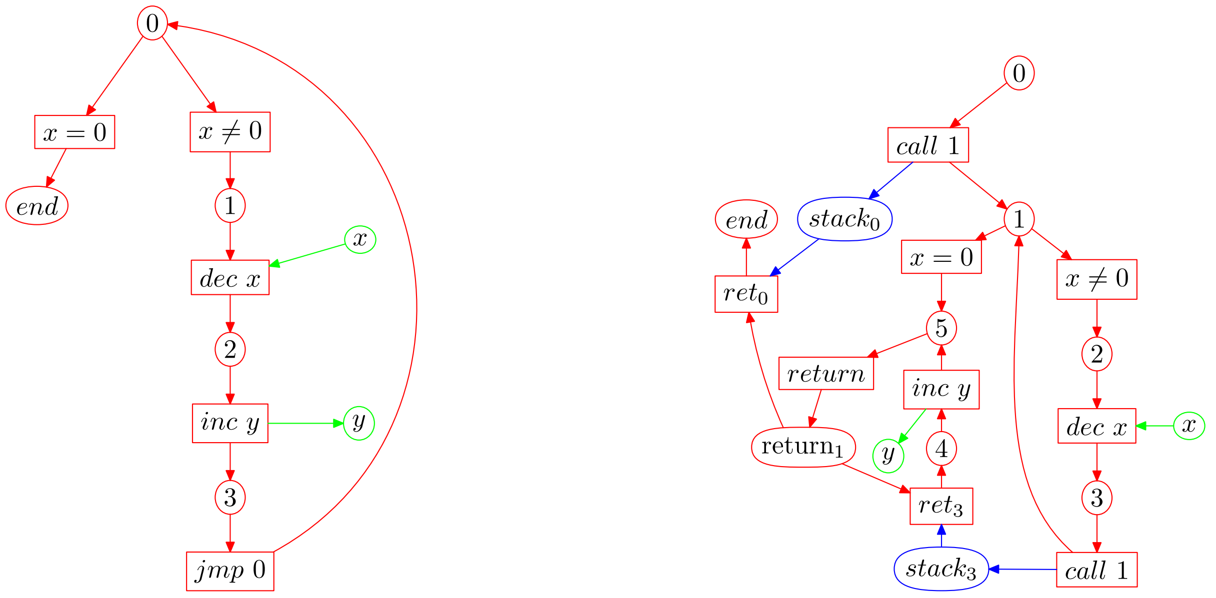

FIG. 7.3 – SCPNs des deux versions de l'addition (à gauche, la version récursive terminale).

Exemple 18.

1. **Addition**.
   Les deux programmes suivants calculent l'addition de deux entiers. Le premier (à gauche) n'utilise pas de `call` et est similaire à la version récursive terminale en langage fonctionnel (c'est-à-dire avec une règle de la forme $\texttt{add}(\mathbf{S}(x), y) \rightarrow \texttt{add}(x, \mathbf{S}(y))$). Le second utilise un `call` et est similaire à la version sans récursion terminale ($\texttt{add}(\mathbf{S}(x), y) \rightarrow \mathbf{S}(\texttt{add}(x, y))$). La deuxième version est obligée "d'emballer" le premier `call` pour éviter un problème si un `return` est rencontré alors que la pile de retour est vide. Le label de fin (*end*) a été déplacé afin de bien séparé le programme de la sous-routine.
   Entrées : x, y.
   Sortie : y.

0	:	jz x *end*	0	:	call 1
1	:	dec x	*end*		
2	:	inc y	1	:	jz x 5
3	:	jmp 0	2	:	dec x
end			3	:	call 1
			4	:	inc y
			5	:	return

   Les SCPNs de ces deux programmes sont présentés sur la figure 7.3.

2. **Copies**.
   Voici maintenant deux macros de copie, *copy* et *dup*. La première, *copy*, recopie le contenu d'une variable dans une autre et met la première variable à 0. La deuxième, *dup* duplique le contenu

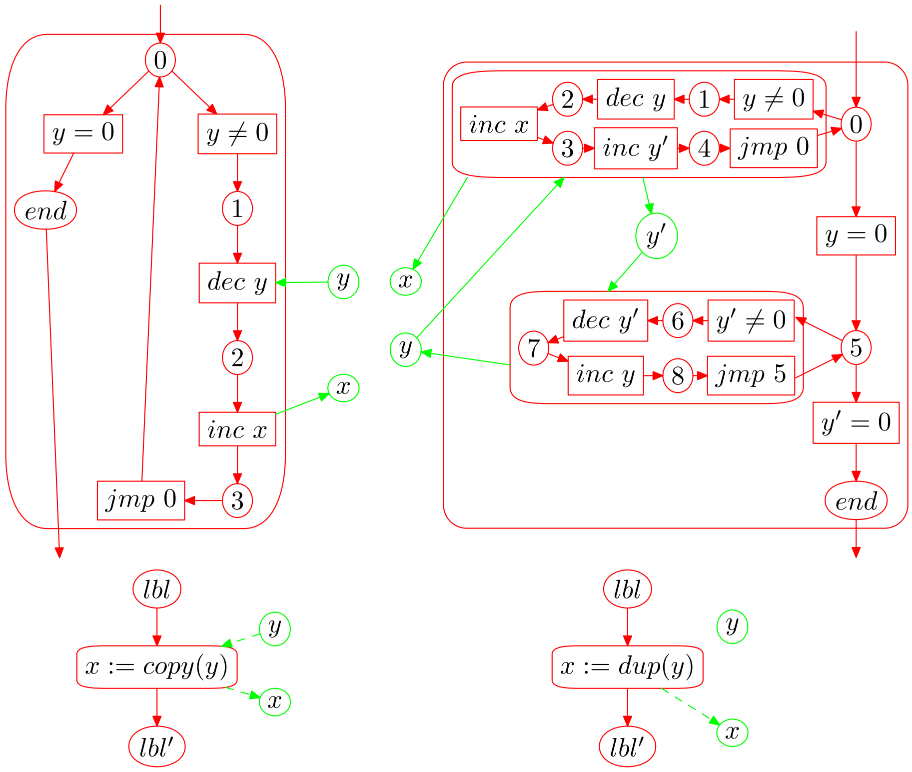

FIG. 7.4 – SCPNs de *copy* et *dup* et versions condensées.

d'une variable dans une autre sans modifier la première variable.

$x := copy(y)$			$x := dup(y)$		
0	:	`jz` y end	0	:	`jz` y 5
1	:	`dec` y	1	:	`dec` y
2	:	`inc` x	2	:	`inc` x
3	:	`jmp` 0	3	:	`inc` y'
end			4	:	`jmp` 0
			5	:	`jz` y' end
			6	:	`dec` y'
			7	:	`inc` y
			8	:	`jmp` 5
			end		

Les SCPNs de ces deux macros sont présentés sur la figure 7.4, accompagnés d'une version "condensée" qui sera utilisée par la suite pour représenter ces macros. Pour le SCPN de *dup*, les deux "boîtes" internes ont été considérées comme chacune une transition afin de pouvoir dessiner un réseau plus clair (notamment au niveau des effets sur les variables).

3. **Plus Grand Commun Diviseur.**
Le programme suivant calcule le pgcd de deux nombres suivant l'algorithme d'Euclide. Par souci de simplification, le test $a < b$ est considéré comme faisant partie du langage. Il est facile de voir qu'on pourrait le représenter sans difficulté à l'aide d'une sous-routine. La macro de copie est utilisée et devrait en toute rigueur être remplacée par l'ensemble des instructions la composant.

Si $a \geq b$ alors $\texttt{pgcd}(a, b) = \texttt{pgcd}(a - b, b)$. Sinon, $\texttt{pgcd}(a, b) = \texttt{pgcd}(b - a, a)$.
Entrées : a, b.
Sortie : a.

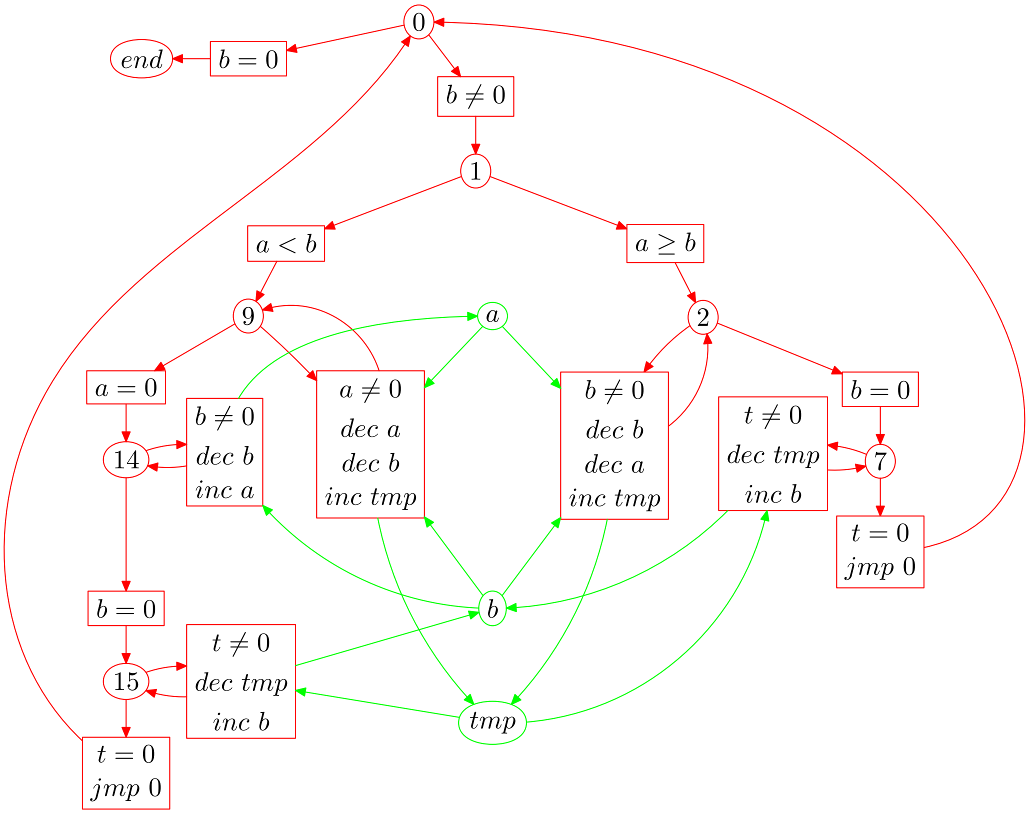

FIG. 7.5 – SCPN simplifié de l'algorithme d'Euclide.

```
 0  :  jz b end
 1  :  if a < b jmp 9
 2  :  jz b 7
 3  :  dec b
 4  :  dec a
 5  :  inc tmp
 6  :  jmp 2
 7  :  b := copy(tmp)
 8  :  jmp 0
 9  :  jz a 14
10  :  dec a
11  :  dec b
12  :  inc tmp
13  :  jmp 9
14  :  a := copy(b)
15  :  b := copy(tmp)
16  :  jmp 0
end
```

Le SCPN de ce programme est montré à la figure 7.5. Par souci de lisibilité, il a été quelque peu simplifié graphiquement (en regroupant dans une même transition plusieurs instructions). Cette simplification n'a aucune conséquence sur le comportement global du réseau.

Maintenant, on définit une fonction qui permet de transformer les états des machines en marquages du SCPN. Ainsi, l'exécution d'un programme, qui correspond à une suite d'états, correspondra à une suite de marquages, donc à une exécution du SCPN.

Définition 95 (États et marquages).
Soit `pgm` un programme. Un état $s = (\mathtt{lbl}, \sigma, \mathtt{stk})$ de `pgm` détermine un marquage $\mathtt{M}(\mathtt{lbl}, \sigma, \mathtt{stk})$ du SCPN.

$$\begin{aligned}
\mathtt{M}(\mathtt{lbl}, \sigma, \mathtt{stk})(\mathtt{lbl}) &= 1 \\
\mathtt{M}(\mathtt{lbl}, \sigma, \mathtt{stk})(\mathtt{lbl}') &= 0 \qquad \text{si } \mathtt{lbl} \neq \mathtt{lbl}'. \\
\mathtt{M}(\mathtt{lbl}, \sigma, \mathtt{stk})(x) &= \sigma(x) \\
\mathtt{M}(\mathtt{lbl}, \sigma, \mathtt{stk})(stack_{\mathtt{lbl}'}) &= \#_{\mathtt{next}(\mathtt{lbl}')}(\mathtt{stk})
\end{aligned}$$

Où $\#_{\mathtt{lbl}}(\mathtt{stk})$ est le nombre d'occurrences de `lbl` dans `stk`.

En fait, $\mathtt{M}(\mathtt{lbl}, \sigma, \mathtt{stk})$ place un jeton dans le label qui correspond à la prochaine instruction à exécuter (c'est le compteur ordinal), autant de jetons dans $stack_{\mathtt{lbl}}$ qu'il y a d'appels de sous-routines depuis `lbl` qui n'ont pas encore terminé et autant de jetons dans chaque variable que la valeur de cette variable.

7.2.2 Terminaison par ressources

Le SCPN d'une machine à compteurs la simule de manière non-déterministe (puisque les sauts conditionnels sont remplacés par des choix non-déterministes entre deux transitions à tirer). Il est maintenant possible de regarder de manière globale le nombre de jetons dans les places "variables" et d'en déduire des propriétés sur la machine.

Définition 96 (Trace étendue).
Soient `pgm` une machine à compteurs, s un état de `pgm` et ω une trace du programme à partir de l'état s. La *trace étendue* à partir de s, notée $\varphi\omega$, est construite à partir de ω de la manière suivante :
- Si ω contient autant de `call` que de $ret_{\mathtt{lbl}}$ alors $\varphi(\omega) = \omega$.
- Si $\omega = \omega_1$ `call lbl` ω_2 `return` ω_3 avec `call lbl` $=$ $\mathtt{instr}(\mathtt{lbl}')$ et autant de `call` que de $ret_{\mathtt{lbl}''}$ dans ω_2 alors $\varphi(\omega) = \varphi(\omega_1$ `call lbl` ω_2 `return` $ret_{\mathtt{lbl}'}\ \omega_3)$.

Théorème 48 (J.-Y. Marion et J.-Y. Moyen, 2003 **[MM03]).**
Soient `pgm` une machine à compteurs et N son SCPN. Soient s un état du programme et M un marquage de N tel que $M \trianglerighteq \mathtt{M}(s)$. Si ω est une trace du programme à partir de l'état s alors $\varphi(\omega)$ est une séquence de tirs à partir de M.

Si ω est finie et que $s \xrightarrow{\omega} s'$ alors il existe un marquage M' de N tel que $M \xrightarrow{\varphi(\omega)} M'$ et $M' \trianglerighteq \mathtt{M}(s')$.

Démonstration.
Par construction du SCPN et par induction sur la longueur de ω. □

Bien sûr, il existe aussi d'autres séquences de tirs qui ne correspondent à aucune exécution du programme. C'est dû au fait que la simulation de la machine à compteurs par le réseau de Petri est faite de manière non-déterministe.

Corollaire 49 (J.-Y. Marion et J.-Y. Moyen, 2003 **[MM03]).**
Si le programme ne termine pas sur toutes ses entrées, alors il existe une séquence de tirs du SCPN infinie.
*Soient `pgm` une machine à compteurs et N son SCPN. Si N termine (*ie *ne comporte aucune exécution infinie), alors `pgm` termine sur toutes ses entrées.*

La terminaison du SCPN implique la terminaison du programme.

Décider si un réseau de Petri termine est dans PTIME. Ça revient à savoir si un système d'inéquations admet des solutions dans $\mathbb{R}$.

En fait, il faut affiner un peu le résultat. En effet, certaines séquences de tirs infinies sont en fait tirables uniquement depuis des marquages qui ne correspondent pas à des marquages initiaux (voire l'exemple du *dup* ci-dessous). Il faut donc étudier la terminaison du réseau *pour un ensemble de marquages initiaux* (globalement un jeton dans la place 0 et un nombre quelconque de jetons dans les places variables d'entrées).

De même, le non-déterminisme de la simulation amène parfois des séquences de tirs infinies qui ne correspondent à aucune exécution du programme. En un sens, ceci est souhaitable (en effet, comme la halte des programmes est indécidable mais qu'on veut une caractérisation décidable, il y aura nécessairement des programmes qui terminent sans qu'on le détecte). Mais si ça arrive trop souvent, ça devient nuisible. Souvent (voir par exemple l'algorithme d'Euclide), ces séquences de tirs correspondent à tirer à la suite une transition "$x = 0$" et une transition "$x \neq 0$", chose qui est bien sûr impossible dans le programme si la valeur de x n'est pas modifiée entre temps. Ces cas peuvent être exclus.

Exemple 19.

1. Le SCPN de l'addition (version non récursive terminale), tel que représenté figure 7.3 termine toujours. En effet, la seule boucle existante dans le réseau est celle qui passe par les places $0, 1, 2, 3$ et elle ne peut pas être tirée une infinité de fois car la place x ne peut contenir qu'un nombre fini de jetons. Donc le programme termine toujours.

   De même, le SCPN de l'addition, version récursive terminale, présenté figure 7.3 termine toujours. En effet, aucun cycle contenant la transition `dec` x (donc les places 2 et 3) ne peut être tiré une infinité de fois (car il y a un nombre fini de jetons dans la place x). Le seul cycle ne contenant pas cette transition est celui qui passe par les places `return`$, 4, 5$. Mais pour le tirer, et plus particulièrement pour tirer la transition ret_3, il faut des jetons dans $stack_3$, ce qui implique d'avoir auparavant tiré `call` 1 et donc aussi `dec` x. Donc le programme termine toujours.
2. Le SCPN de la copie, présenté figure 7.4 termine toujours. En effet, la boucle qui passe par les places $0, 1, 2, 3$ ne peut être tirée infiniment sans vider la place y. Donc le programme termine toujours.

   Le SCPN de la macro *dup*, présenté figure 7.4 ne termine pas toujours. En effet, si on a un jeton dans chacune des places $0, 5, y, y'$ on peut tirer alternativement les transitions de la boucle du haut et celles de la boucle du bas et revenir ainsi à l'état initial. Cependant, cette exécution infinie nécessite deux jetons dans des places ordinales. Or, dans l'état initial du réseau, il n'y a qu'un jeton dans ces places (il se trouve dans la place 0). Comme le nombre de jetons dans les places ordinales est conservé, cette exécution infinie ne correspond à aucun état initial du système. Donc le SCPN termine depuis tout état initial, donc le programme termine toujours.
3. Le SCPN de l'algorithme d'Euclide, présenté à la figure 7.5 ne termine pas. En effet, il existe une exécution infinie avec un seul jeton ordinal qui passe par les places $0, 1, 2, 7$. Cependant, cette exécution infinie tire successivement une transition $b \neq 0$ et une transition $b = 0$ sans modifier entre temps la valeur de b et est donc incohérente. Si on se restreint aux exécutions cohérentes, le SCPN n'a pas d'exécution infinie, donc le programme termine toujours.

Si on regarde ces exemples, on constate qu'on a trois problèmes à examiner (simultanément ou successivement) :
- La terminaison du réseau de Petri en tant que tel.
- La terminaison depuis un état initial, ou la validité d'une séquence de tirs infinie vis-à-vis des conditions sur les places ordinales.
- La terminaison en se restreignant aux séquences de tirs cohérentes vis-à-vis des variables.

Théorème 50.
Soient `pgm` *une machine à compteurs et N son SCPN. Si N n'admet aucune exécution infinie cohérente vis-à-vis des variables et qui nécessite un seul jeton ordinal, alors* `pgm` *termine sur toutes ses entrées.*

Théorème 51.
Soient `pgm` *une machine à compteurs et N son SCPN. Savoir si il existe une exécution infinie de N qui nécessite un seul jeton dans les places ordinales et qui soit cohérente vis-à-vis des variables est décidable et dans* NPTIME.

Ça revient à savoir si un système d'inéquations linéaires admet une solution.

Idées pour la preuve.
La preuve complète de ce théorème est donnée au dernier chapitre. La construction précise des matrices à utiliser dans le système est assez fastidieuse et l'exposer ici nuirait à la compréhension globale de la méthode.

Les idées directrices pour cette preuve sont les suivantes :

- La terminaison simple du réseau de Petri se fait très simplement en cherchant une séquence de tirs qui soit une boucle, cette séquence de tir étant décrite uniquement par son vecteur caractéristique.
- L'unicité du jeton ordinal pour une séquence de tirs donnée peut se trouver (c'est le problème du marquage initial minimal). Malheureusement, on a ici plusieurs séquences de tirs à tester et on ne peut pas appliquer les solutions usuelles de ce problème. On teste donc si le réseau extrait en ne conservant que les transitions tirées et les places ordinales s'y rattachant est connexe. Ce test peut se faire par un calcul de flot dans un graphe.
- La cohérence de la séquence de tirs peut s'obtenir de deux manières. La première est d'empêcher $x = 0$ et $x \neq 0$ d'être tirées si `inc` x et `dec` x ne sont pas tirées. la deuxième consiste à déplier un peu plus le SCPN de manière à ce que deux transitions donnant des informations incohérentes sur une même variable ne puissent pas être tirées sans avoir modifié la valeur de la variable. La première solution s'exprime facilement sous forme d'inéquations linéaires. La deuxième s'apparente à la déforestation de P. Wadler [Wad90] et est détaillée ci-dessous.

□

7.3 Réseau avec contraintes

Dans le cas de l'algorithme d'Euclide, la non-terminaison provient d'une exécution infinie qui tire les transitions $b \neq 0$ et $b = 0$ à la suite, sans modifier entre temps la valeur de b. J'ai parlé de deux solutions pour éliminer ces cas. La première consiste à ajouter au système d'inéquations des contraintes (c'est-à-dire de nouvelles inéquations) qui empêchent une telle exécution d'être solution du système global. Cela revient à placer un peu de sémantique (une variable ne peut pas avoir deux valeurs différentes en même temps) dans le système. L'autre solution consiste à placer la sémantique dans la construction du réseau.

On construit donc un nouveau réseau, où à chaque place correspondant à un label est associé un ensemble de contraintes de la forme $x = 0$ ou $x \neq 0$. Les contraintes sont associées aux places de manière très simple : toutes les places situées "entre" une transition $x = 0$ et une transition `inc` x ont la contrainte $x = 0$ et toutes les places situées entre $x \neq 0$ et `dec` x ont la contrainte $x \neq 0$. Les places qui devraient apparaître avec des ensembles de contraintes différentes sont dupliquées de manière à ne pas être confondues. Les transitions qui créeraient un ensemble de contraintes non cohérent (c'est-à-dire contenant à la fois $x = 0$ et $x \neq 0$) ne sont pas représentées (ainsi que toute la partie du réseau qui leur est rattachée). La place 0 initiale a les contraintes $x = 0$ pour tout x qui n'est pas une variable d'entrée.

Par exemple, dans le cas de l'algorithme d'Euclide, la place 1 a les contraintes $tmp = 0$ (héritée de 0) et $b \neq 0$ (car elle est après le test). De même pour la place 2. Mais du coup, la transition $b = 0$ n'apparaît plus dans $2^{\bullet}_{b\neq 0, tmp\neq 0}$ car la nouvelle contrainte ($b = 0$) est incompatible avec l'ancienne ($b \neq 0$). Il n'y a donc qu'une seule transition ($b \neq 0$) à cet endroit. Mais sitôt après avoir tiré `dec` b, la contrainte $b \neq 0$ est levée (de même la contrainte $tmp = 0$ est levée après avoir tiré `inc` tmp). On se retrouve donc de nouveau au label 2, mais cette fois sans aucune contrainte. Il s'agit donc d'une nouvelle place, différente de l'ancienne, de laquelle les deux transitions peuvent être tirées.

Une partie du réseau ainsi obtenu (celle correspondant à la branche $a \geq b$) est dessinée sur la figure 7.6. Les places correspondant à des variables ont été omises pour plus de clarté.

Sur ce nouveau réseau, on voit tout de suite que la séquence de tirs infinie précédente n'est plus possible. Et que la boucle qui est représentée ici ne peut plus être prise indéfiniment car elle vide la place a. Si on trace le réseau complet, on peut voir qu'il n'a aucune séquence de tirs infinie. Donc le programme termine toujours.

L'inconvénient principal de cette nouvelle méthode est évident : la taille du réseau peut beaucoup augmenter (jusqu'à être multipliée par un facteur 3^k où k est le nombre de compteurs de la machine). Mais elle permet de montrer la terminaison de plus de systèmes.

En outre, cet inconvénient donne une idée de méthode plus "progressive" pour tenter de prouver la terminaison : on commence par essayer avec le SCPN normal, puis si ça ne marche pas on ajoute des contraintes et on recommence. On peut rajouter plus de contraintes que simplement celles introduites par les tests. En effet, après un `inc` x on sait que la contrainte $x \neq 0$ est vraie.

On peut même pousser le raisonnement plus loin. Pourquoi enlever complètement la contrainte $x = 0$ après un `inc` x ? On connaît à ce moment la valeur précise de x et on pourrait tout à fait mettre la

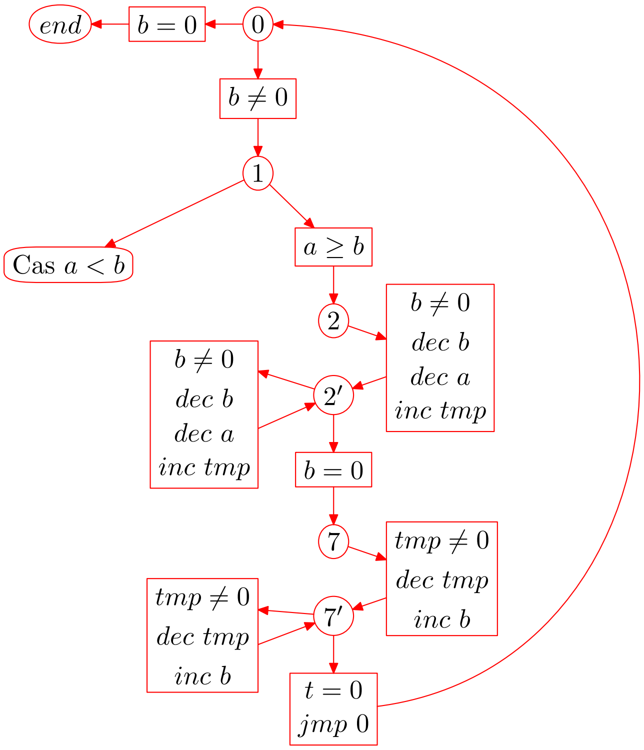

FIG. 7.6 – Une partie du CFPN avec contraintes de l'algorithme d'Euclide.

contrainte $x = 1$ à cet endroit. Et continuer ainsi tant que les valeurs des variables sont connues. De même, après un `inc` x on peut ajouter la contrainte $x > 0$, puis $x > 1$, ...

Mais si on continue trop longtemps, un problème majeur risque d'arriver : la taille du réseau de Petri n'est plus bornée ni même nécessairement finie, autrement dit l'analyse elle même peut ne pas terminer ! Ce qui est sans aucun doute la dernière chose souhaitable.

D'autre part, la construction du CFPN avec contrainte peut permettre d'éliminer certains calculs inutiles. Ainsi, on peut détecter que les instructions

$$\begin{array}{lcl} 0 & : & \texttt{inc}\ x \\ 1 & : & \texttt{jz}\ x\ \texttt{lbl} \\ 2 & : & \texttt{dec}\ x \end{array}$$

sont en fait inutiles (le CFPN avec contraintes n'a pas de branchement au label 1 et les instructions `inc` et `dec` se retrouvent donc à la suite l'une de l'autre). Et on peut à partir du SCPN avec contraintes (simplifié) réécrire un programme correspondant, lui aussi simplifié et donc plus efficace (moins de tests inutiles).

Cette construction, cette simplification éventuelle, et même le problème de non-terminaison de la transformation, sont très similaires à ce que fait P. Wadler avec la déforestation [Wad90]. Je pense qu'il est effectivement possible d'obtenir des résultats similaires (c'est-à-dire d'éviter de construire des structures de données intermédiaires) en plaçant des contraintes dans les CFPN.

Chapitre 8

Calcul en place

J'essaie maintenant de détecter les programmes "Non-Size Increasing" (NSI), c'est-à-dire qui calculent en place. De manière informelle, cela veut dire qu'aucune mémoire supplémentaire n'est nécessaire pour faire le calcul. L'addition, ou le calcul du pgcd sont calculés par des programmes NSI. Au début du calcul, une certaine quantité de mémoire, donnée par la taille des variables (plus une constante) est allouée et au cours du calcul, le programme n'utilise jamais plus de mémoire.

Les programmes NSI ont été introduits par M. Hofmann [Hof99, Hof02] qui a montré que de tels programmes pouvaient être écrits en `C` sans utiliser l'instruction `malloc`, c'est-à-dire sans allouer de mémoire supplémentaire, explicitant ainsi la réutilisation de la mémoire (des pointeurs) par ces programmes.

Définition 97 (NSI).
Soit `pgm` une machine à compteurs. `pgm` est NSI (Non-Size Increasing) si il existe une constante c telle que pour toutes les entrées $v_1, \ldots, v_m$, si $\texttt{pgm} \vdash 0, \{i_1 \leftarrow v_1, \ldots, i_m \leftarrow v_m\}, [] \xrightarrow{*} \texttt{lbl}, \sigma, \texttt{stk}$, on a $\sum_x |\sigma(x)| \leq \sum_{i=1}^{m} |v_i| + c$.

8.1 Resource Petri Net

Le RPN (Resource Petri Net) d'une machine à compteurs est similaire au SCPN avec une place supplémentaire qui représente la mémoire. Les RPN des instructions de base sont représentés sur la figure 8.1 et servent de définition informelle. Le nombre de jetons dans la place mem représente les ressources libres et on peut les voir comme des pointeurs vers des emplacements libres de la mémoire. À chaque fois qu'une instruction `dec` est exécutée, on met un jeton en mémoire, à chaque fois qu'une instruction `inc` est exécutée, on enlève un jeton de la mémoire. Les tests $x \neq 0$ sont aussi protégés : il est maintenant nécessaire d'avoir un jeton dans la place x (c'est-à-dire d'avoir réellement $x \neq 0$) pour pouvoir tirer la transition correspondante, mais le contenu de cette place (donc la valeur de x) n'est pas modifié. Cette dernière condition permet d'éviter des blocages du réseau de Petri sur un `dec`, c'est-à-dire au milieu du programme.

Définition 98 (RPN).
Soient `pgm` une machine à compteurs et $N = (S, T, F)$ son SCPN. Le RPN $N' = (S', T', F')$ de `pgm` est défini par :
- $S' = S \bigcup \{mem\}$.
- $T' = T$.
- $F \subset F'$

Et on ajoute en outre dans F' les arcs nécessaires pour avoir :
- $mem \in^{\bullet} (\texttt{inc}\ x)$ pour toute instruction $\texttt{inc}\ x$ de `pgm`.
- $mem \in (\texttt{dec}\ x)^{\bullet}$ pour toute instruction $\texttt{dec}\ x$ de `pgm`.
- $x \in^{\bullet} (x \neq 0)$ et $x \in (x \neq 0)^{\bullet}$ pour toute transition $x \neq 0$ de N'.

Tout ce qui se rapporte à la mémoire (*ie* la place mem et les arcs qui y arrivent ou en partent) est représenté en bleu sur les schémas.

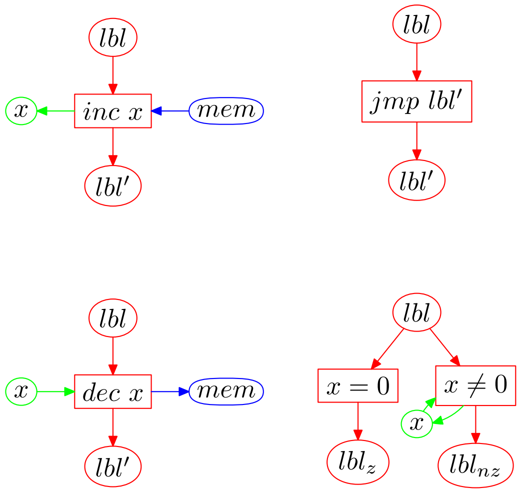

Fig. 8.1 – Resource Petri net des instructions.

Exemple 20.
Le RPN de l'addition (version récursive terminale) est dessiné dans la figure 8.2.

Lemme 52 (S-invariant).
Soient `pgm` *une machine à compteurs et N son RPN. Le nombre de jetons contenu dans les places variables (x) et mémoire (mem) est constant lors de toute exécution de N.*

Démonstration.
Chaque transition a exactement le même nombre de telles places en entrée et en sortie.

Effectivement, les seules transitions qui ont une place variable en entrée ou en sortie sont
- `dec` x qui a x en entrée et mem en sortie.
- `inc` x qui a x en sortie et mem en entrée.
- $x \neq 0$ qui a x en entrée et en sortie.

□

Remarque :
En fait, les SCPN gèrent deux types de ressources qui sont séparées. D'une part les valeurs des variables, représentées par des jetons dans les places variables, et d'autre part les piles de retour, représentées par des jetons dans des places $stack_{\texttt{lbl}}$. Ces deux types de ressources correspondent respectivement au tas (heap) et à la pile (stack) dans les langages de programmation usuels. Ici, la place mémoire ne contrôle que les variables, c'est-à-dire la mémoire utilisée par le tas. Il est très facile de faire de même avec la pile (en mettant mem dans le préset des `call` et dans le postset des $ret_{\texttt{lbl}}$). On pourrait alors contrôler l'utilisation de mémoire soit sur la pile, soit sur le tas. Ceci est particulièrement utile si on pense à des algorithmes récursifs comme le tri rapide qui a besoin d'une pile non bornée (de taille logarithmique par rapport à la taille de la liste à trier) mais n'a pas besoin de mémoire supplémentaire sur le tas.

On définit maintenant des marquages initiaux et finaux qui correspondent au début ou à la fin de l'exécution d'un programme. Un marquage initial place le compteur ordinal dans la place 0, des jetons dans les places d'entrées et dans mem. Un marquage final place le compteur ordinal dans la place end et des jetons dans les places non ordinales.

Définition 99 (Marquages initiaux et finaux).
Un marquage M est *initial* si il existe une substitution σ_i telle que $\sigma_i(x) = 0$ pour chaque variable qui n'est pas une variable d'entrée et $M(p) = \mathtt{M}(\mathtt{main}, \sigma_i, [])(p)$ pour chaque place p différente de mem.

Un marquage M est *final* si il existe une substitution σ_f telle que $M(p) = \mathtt{M}(end, \sigma_f, [])(p)$ pour chaque place p différente de mem.

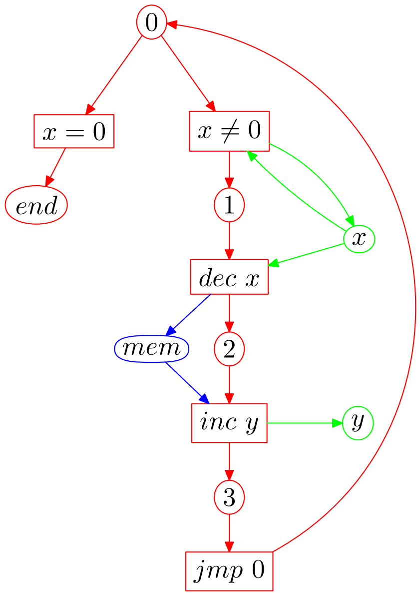

FIG. 8.2 – RPN de l'addition.

Lemme 53.
Un marquage final est une impasse.

Démonstration.
Chaque transition a une place ordinale différente de *end* parmi ses places d'entrées. □

8.2 Détecter les programmes Non-Size Increasing

Définition 100.
Soient pgm une machine à compteurs et N son RPN. N est *sans allocation de mémoire* si il existe une constante c telle que pour chaque marquage initial M_i qui place c jetons dans *mem* ($M_i(mem) = c$) et pour chaque séquence de tirs maximale (pour la longueur) ω on ait $M_i \xrightarrow{\omega} M_f$ avec M_f un marquage final.

Théorème 54.
Soient pgm *une machine à compteurs et N son RPN.* pgm *est NSI si N est sans allocation de mémoire.*

Démonstration.
Supposons que pgm ne soit pas NSI. Soient c un entier quelconque et σ_i une substitution des variables d'entrées (c'est à dire que $\sigma_i(x) = 0$ si x n'est pas une variable d'entrée). Soient $s = (\mathtt{lbl}, \sigma, \mathtt{stk})$ et $s' = (\mathtt{lbl}', \sigma', \mathtt{stk}')$ deux états de la machine à compteurs tels que

- $(0, \sigma_i, []) \xrightarrow{\omega} s$
- $s \xrightarrow{(\mathtt{inc}\ x)} s'$
- $\sum_x \sigma'(x) > \sum_x \sigma(x) = \sum_x \sigma_i(x) + c$.

De tels états et un telle trace ω existent si pgm n'est pas NSI. On pose $\omega' = \omega(\mathtt{inc}\ x)$.

Par construction, $\varphi(\omega')$ est une exécution du SCPN de pgm qui est activée par n'importe quel marquage $M \trianglerighteq \mathtt{M}(0, \sigma_i, [])$ et en particulier pour $M = \mathtt{M}(0, \sigma_i, [])$.

Soient N le RPN de pgm et M'_0 un marquage de N tel que $M'_0(p) = \mathtt{M}(0, \sigma_i, [])$ pour toute place p différente de *mem*. M'_0 est un marquage initial de N.

Si $\varphi(\omega')$ est une séquence de tirs de N activée par M'_0, il existe un marquage M' tel que $M'_0 \xrightarrow{\omega'} M'$. Par construction, on a $M' \trianglerighteq \mathtt{M}(s')$ donc

$$\sum_x M'(x) = \sum_x \sigma'(x) > \sum_x \sigma_i(x) + c$$

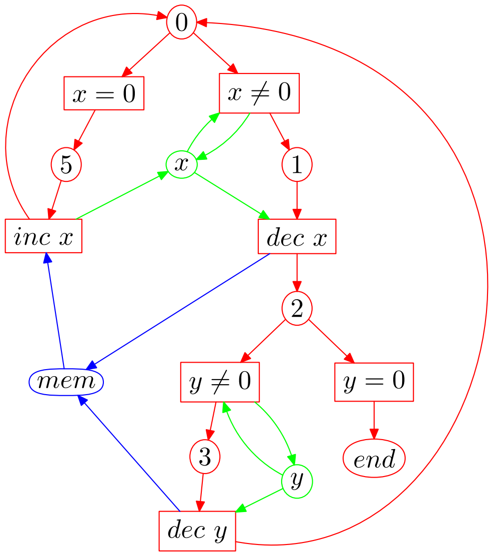

FIG. 8.3 – Le RPN d'un programme NSI n'est pas forcément sans allocation de mémoire.

Par application du S-invariant sur les variables et la mémoire, on a

$$\sum_x M'(x) + M'(mem) = \sum_x M'_0(x) + M'_0(mem)$$

Et par construction de M'_0, on a $\sum_x M'_0(x) = \sum_x \sigma_i(x)$.

On peut donc en déduire $M'_0(mem) > c$.

Ainsi, si `pgm` n'est pas NSI, quel que soit c on peut construire une séquence de tirs qui nécessite plus de c jetons dans mem pour aboutir à un état final. Donc le RPN de `pgm` n'est pas sans allocation de mémoire.

En contraposant, si le RPN de `pgm` est sans allocation de mémoire, `pgm` est NSI. □

Exemple 21.
Le RPN de l'addition donné à la figure 8.2 est sans allocation de mémoire. En effet, on ne peut pas enlever de jeton de mem sans en avoir ajouté avant. Ainsi, quel que soit le nombre de jetons dans mem au début, une exécution partant d'un état initial termine toujours dans un état final (`dec` x ne peut pas causer d'impasse car les arcs rajoutés vers la transition $x \neq 0$ assure qu'il y a au moins un jeton dans x avant de prendre cette branche).

Donc l'addition est Non-Size Increasing.

Remarque :
Malheureusement la réciproque n'est pas aussi directe que ça. Le programme suivant par exemple est NSI mais son RPN (dessiné sur la figure 8.3) n'est pas sans allocation de mémoire à cause de la boucle entre les places 0 et 5.

entrées : x, y

sorties : y

0	:	`jz` x 5
1	:	`dec` x
2	:	`jz` y end
3	:	`dec` y
4	:	`jmp` 0
5	:	`inc` x
6	:	`jmp` 0
end		

On peut cependant remarquer que cette boucle tire une transition $x = 0$ juste après avoir tiré une

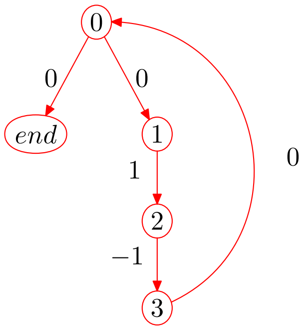

FIG. 8.4 – Graphe correspondant au RPN de l'addition.

transition `inc` x. En construisant le réseau avec contraintes comme au chapitre précédent, on pourrait donc détecter que ce programme est NSI. Il est probable que sur les réseaux avec contraintes, c'est-à-dire en ne s'occupant que des exécutions qui sont cohérentes vis-à-vis des variables, la réciproque soit vraie.

Théorème 55.
Soient `pgm` une machine à compteurs, S son SCPN et N son RPN. Si S termine (éventuellement à l'aide des conditions d'unicité du compteur ordinal et de cohérence des tirs), on peut décider si N est sans allocation de mémoire en temps polynômial dans la taille de N (donc dans la taille de `pgm`).

Démonstration.
On construit un graphe orienté $G = (V, E)$ à partir de N. V est l'ensemble des places ordinales et il y a un arc entre u et v si il y a une transition entre les places correspondantes. Sur chaque arc est placé un poids qui vaut -1 si la transition correspondante est `inc`, 1 si la transition est `dec` et 0 sinon.

Une exécution de N (avec un seul jeton ordinal et cohérente vis-à-vis des variables) correspond à un chemin dans G. Le nombre de jetons à placer dans mem initialement est l'opposé du poids de ce chemin (le poids du chemin correspond au nombre de jetons qui ont été ajoutés dans mem). Ainsi, N est sans allocation de mémoire si et seulement si tous les chemins de G qui partent de 0 sont de poids supérieur ou égal à une constante c qui ne dépend que du graphe. Cette dernière condition équivaut à ce qu'il n'y ait pas de boucle de poids négatif dans G.

Pour savoir si il existe une boucle de poids négatif, une méthode est de calculer la clôture transitive de la matrice d'adjacence de G, ce qui peut être fait en temps polynômial ($O(n^4)$). □

Exemple 22.
Le graphe correspondant au RPN de l'addition est dessiné dans la figure 8.4. Comme il ne contient aucun cycle de poids négatif, le RPN de l'addition est bien sans allocation de mémoire, donc l'addition est NSI.

L'existence d'une boucle de poids négatif est aussi équivalente à l'existence d'une solution au système d'inéquations

$$X \times I_G \leq W$$

où I_G est la transposée de la matrice d'incidence de G (une ligne par sommet, une colonne par arc, -1 si l'arc part du sommet, 1 si il y arrive, 0 sinon) et W est le vecteur du poids des arcs. On peut consulter la section 25.5 de [CLR94] pour voir la preuve de cette équivalence.

L'avantage de cette dernière caractérisation est d'être sous forme d'inéquations linéaires. Ainsi, on peut facilement l'ajouter aux autres inéquations obtenues pour savoir si le réseau termine et résoudre toutes ces questions à la fois.

8.3 Quelques exemples supplémentaires

Voici quelques exemples d'autres programmes qui sont analysables par cette méthode. J'explique aussi comment étendre le langage pour traiter les listes.

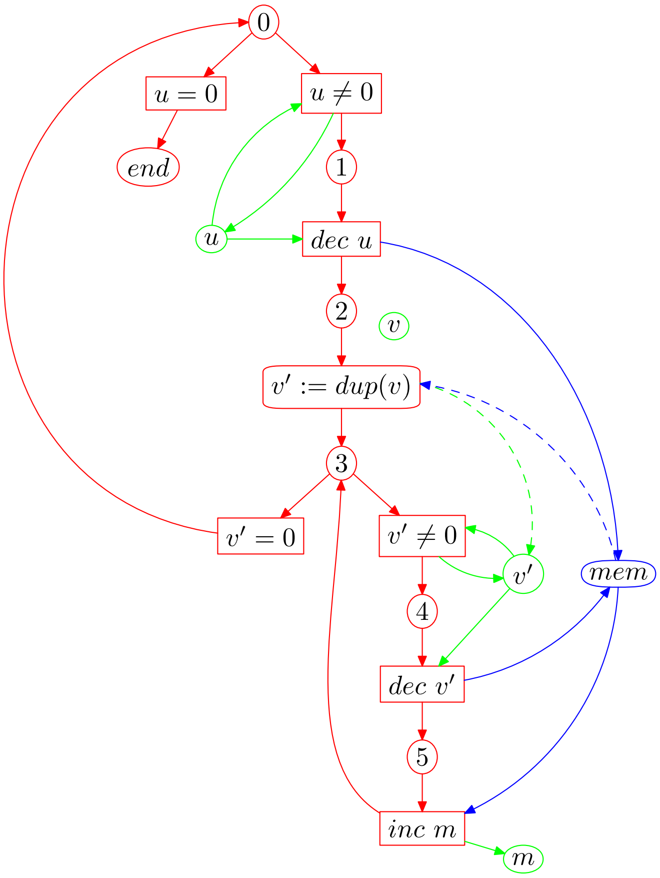

FIG. 8.5 – RPN de la multiplication.

Multiplication

Le programme suivant calcule la multiplication de deux nombres.
Entrées : u, v.
Sortie : m.

$$\begin{array}{rcl} 0 & : & \texttt{jz}\ u\ end \\ 1 & : & \texttt{dec}\ u \\ 2 & : & v' := v \\ 3 & : & \texttt{jz}\ v'\ 0 \\ 4 & : & \texttt{dec}\ v' \\ 5 & : & \texttt{inc}\ m \\ 6 & : & \texttt{jmp}\ 3 \\ end & & \end{array}$$

Le RPN de ce programme est dessiné sur la figure 8.5. Le réseau a été simplifié en regroupant ensemble certaines transitions.

La présence de la macro *dup* cause une boucle infinie. Mais si on impose l'unicité du compteur ordinal, il n'y a plus d'exécution infinie possible du SCPN donc le programme termine toujours.

En revanche, toujours à cause de la macro *dup*, le RPN n'est pas sans allocation de mémoire. En effet, quel que soit le nombre de jetons initialement dans *mem* il existe un marquage qui place suffisamment de jetons dans u et v pour que le réseau se bloque pendant l'exécution de *dup* (qui consomme potentiellement autant de mémoire que la valeur de v), donc dans un état qui n'est pas final.

Tri par insertion

Je veux maintenant écrire un programme qui trie une liste d'entiers. Comme les listes n'existent pas dans les machines à compteurs, il faudrait en toute théorie introduire un codage des listes dans les entiers et travailler uniquement avec des entiers. Ceci donnerait lieu à un programme pour le moins confus. C'est pourquoi j'ai préféré étendre le langage en utilisant des listes. Comme les éléments de la liste n'ont aucune valeur calculatoire (et en particulier ne servent pas à prouver la terminaison à partir du moment où la comparaison termine), on peut les laisser entièrement de côté dans l'analyse. Le SCPN ne comportera donc des places que pour les variables qui sont des listes, et la gestion de la mémoire se fera uniquement sur la longueur des listes, ce qui est d'ailleurs plus proche des vrais ordinateur (avec des listes chaînées) ou de l'approche originelle de M. Hofmann. Cela revient à considérer que les entiers occupent tous la même place en mémoire.

Pour construire et détruire les listes, il faut deux nouvelles instructions, `cons` et `hd`, jouant un rôle similaire à `inc` et `dec`. $\texttt{cons}(a, l)$ prend un entier et une liste et ajoute l'entier en tête de la liste. $a = \texttt{hd}(l)$ enlève le premier élément de la liste et place sa valeur dans a.

De même, le test `jz` est modifié de manière à être utilisable sur les listes. $\texttt{jz}\ l\ \texttt{lbl}$ saute si et seulement si la liste est vide. Finalement, j'ajoute une comparaison entre entiers $if\ a < b\ jmp$ `lbl` qui saute si et seulement si la valeur de a est plus petite que celle de b. Il serait facile d'écrire une macro pour faire ce calcul. Il faudrait alors prouver la terminaison de la macro avant de pouvoir la considérer comme une instruction du langage.

Le programme suivant insère un nombre à la bonne position dans une liste déjà triée. Un accumulateur est nécessaire pour conserver la valeur des éléments du début de la liste et les remettre en place une fois l'insertion effectuée.

Entrées : a, l.

Sortie : l.

$$\begin{array}{rcl}
I0 & : & \texttt{jz}\ l\ I6 \\
I1 & : & b := \texttt{hd}(l) \\
I2 & : & if\ a \leq b\ \texttt{jmp}\ I5 \\
I3 & : & \texttt{cons}(b, acc) \\
I4 & : & \texttt{jmp}\ I0 \\
I5 & : & \texttt{cons}(b, l) \\
I6 & : & \texttt{cons}(a, l) \\
I7 & : & \texttt{jz}\ acc\ Iend \\
I8 & : & b := \texttt{hd}(acc) \\
I9 & : & \texttt{cons}(b, l) \\
I10 & : & \texttt{jmp}\ I7 \\
Iend & &
\end{array}$$

Le RPN de l'insertion est montré sur la partie gauche de la figure 8.6. Il termine toujours car ni la première boucle $(I0, I1, I2, I3)$ ni la deuxième $(I7, I8, I9)$ ne peuvent être tirées infiniment sans vider l ou acc. Il est cependant nécessaire de s'assurer de l'unicité du compteur ordinal, sinon en tirant alternativement chacune des deux boucles on aboutit à une exécution infinie. Il est sans allocation de mémoire car aucune boucle du graphe sous-jacent n'a un poids négatif (en consommation de mémoire) donc le programme est NSI (il faut mettre un jeton dans mem au début de l'exécution).

Le programme suivant calcule le tri proprement dit. Les labels $Iend$ de l'insertion et 0 du tri sont confondus (c'est-à-dire que les occurrences de $Iend$ dans le programme de tri doivent être remplacées par 0). Dans le dessin de la figure 8.6, les deux places sont séparées de manière à avoir deux réseaux séparés.

Entrée : l'.

Sortie : l.

$$\begin{array}{rcl}
0 & : & \texttt{jz}\ l'\ end \\
1 & : & a := \texttt{hd}(l') \\
2 & : & \texttt{jmp}\ I0 \\
end & &
\end{array}$$

Le RPN termine toujours car la boucle $(0, 1, I0, \ldots, Iend)$ ne peut être tirée infiniment sans vider la place l'. Le réseau est sans allocation de mémoire donc le programme est NSI.

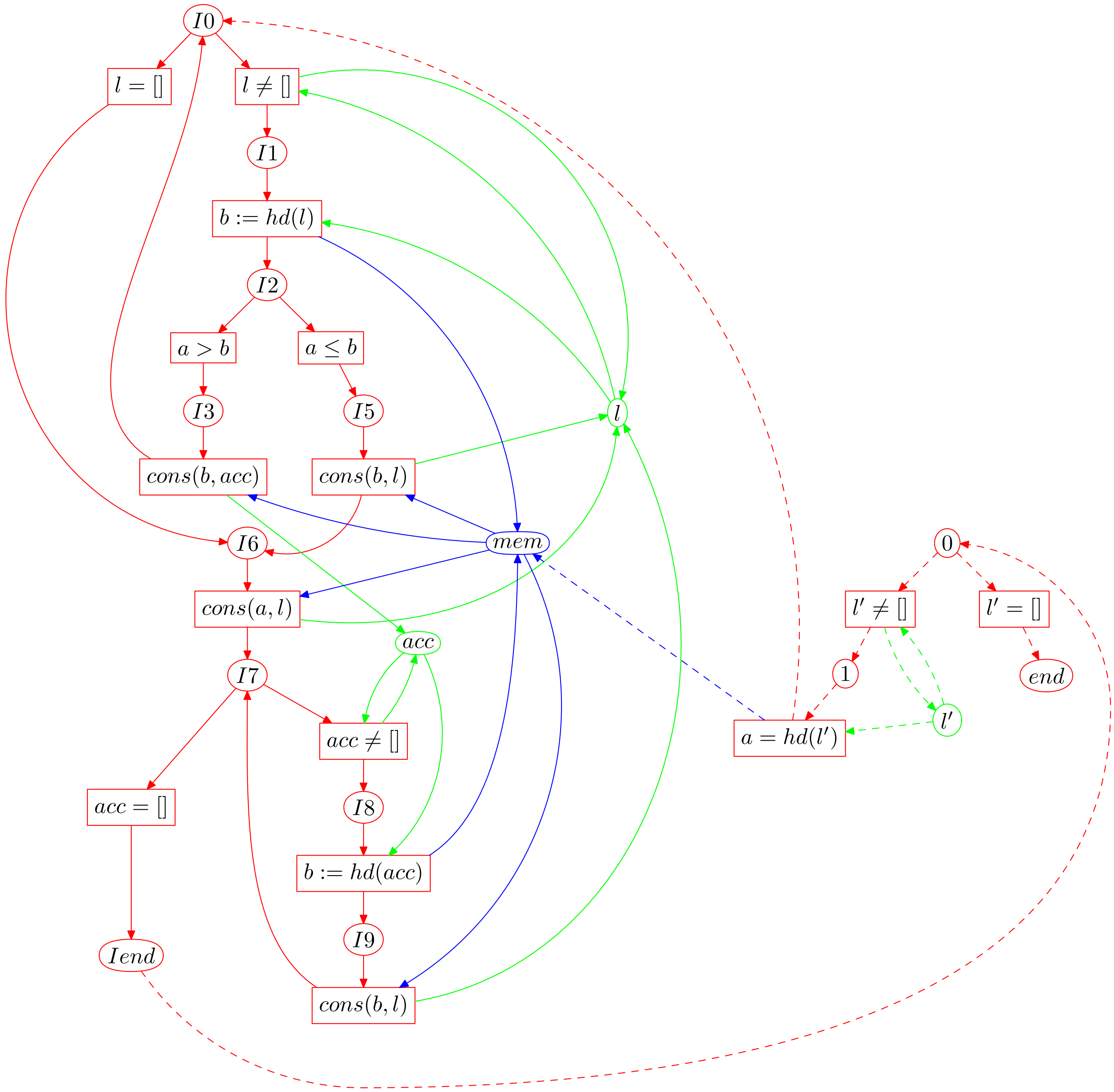

FIG. 8.6 – RPN du tri par insertion.

Tri rapide

Je montre ici que la technique peut aussi s'appliquer au tri rapide et donc, potentiellement, à tous les algorithmes "Diviser pour régner".

De manière à gérer les appels récursifs et la pile de retour des valeurs, il est nécessaire d'introduire encore une extension du langage. La fonction `cons'` a maintenant trois arguments. Le deuxième et le troisième sont les arguments usuels (élément à ajouter et liste). Le premier est un entier naturel qui compte le nombre de *tag*s sur la liste. Une liste qui commence par un `cons'` avec un nombre strictement positif de tags est considérée vide à tous points de vue (en particulier pour un `jz`). La fonction `tag` (resp. `untag`) ajoute (resp. enlève) un tag à la liste, c'est-à-dire incrémente (resp. décrémente) le premier élément du premier `cons'` de la liste. La fonction `hd` renvoie une erreur si elle est appelée sur une liste tagguée (comme si la liste était vide) ; la fonction `cons` crée une liste sans tag sur le premier élément (celui qu'elle ajoute). La taille des listes double (car il y a deux fois plus d'entiers à stocker pour une liste de même longueur), mais comme on ne mesure que la longueur des listes, `tag` et `untag` n'auront aucun effet sur celle-ci et donc aucun effet sur les jetons dans les places variables.

concat

Commençons par écrire une fonction qui concatène deux listes (en ajoutant un élément de plus au milieu).

Entrées : l_1, a, l_2

Sortie : l'

$$\begin{array}{rcl}
C0 & : & \texttt{jz}\ l_1\ C4 \\
C1 & : & b = \texttt{hd}(l_1) \\
C2 & : & \texttt{cons}(b, acc') \\
C3 & : & \texttt{jmp}\ C0 \\
C4 & : & \texttt{cons}(a, acc') \\
C5 & : & \texttt{jz}\ l_2\ C9 \\
C6 & : & b = \texttt{hd}(l_2) \\
C7 & : & \texttt{cons}(b, acc') \\
C8 & : & \texttt{jmp}\ C5 \\
C9 & : & \texttt{jz}\ acc'\ Cend \\
C10 & : & b = \texttt{hd}(acc') \\
C11 & : & \texttt{cons}(b, l') \\
C12 & : & \texttt{jmp}\ C9 \\
Cend & &
\end{array}$$

On peut voir que le programme termine toujours et est NSI.

split

Ensuite une fonction qui sépare une liste en deux en fonction d'un nombre a, d'une part les éléments plus petits que a et d'autre part les éléments plus grands que a.

Entrées : a, l

Sorties : l_1, l_2

$$\begin{array}{rcl}
S0 & : & \texttt{jz}\ l\ Send \\
S1 & : & b = \texttt{hd}(l) \\
S2 & : & if\ a < b\ then\ \texttt{jmp}\ S5 \\
S3 & : & \texttt{cons}(b, l_1) \\
S4 & : & \texttt{jmp}\ S0 \\
S5 & : & \texttt{cons}(b, l_2) \\
S6 & : & \texttt{jmp}\ S0 \\
Send & &
\end{array}$$

Là encore, on peut voir que la fonction termine et est NSI.

sort

Finalement, voici l'algorithme de tri proprement dit. Les tags sont utilisés ici pour conserver les résultats précédents ou les données non encore traitées, hors de portée du traitement courant mais à l'intérieur des variables utilisées. On arrive ainsi à simuler une pile dans laquelle serait stockée ces données. Les quatre utilisations de la macro *copy* doivent en fait être développées en quatre morceaux de programme différents car elles n'ont pas lieu entre les mêmes variables et on ne peut donc pas réutiliser le même bout de programme.

Le programme complet, avec toutes les macros déroulées et les sous-routines placées ensemble contient 70 instructions. Un tel nombre s'explique aisément par le manque d'expressivité du langage.

Entrée : l

Sortie : l'

$$\begin{array}{rcl}
0 & : & \texttt{jz}\ l\ 16 \\
1 & : & a = \texttt{hd}(l) \\
2 & : & \texttt{cons}(a, acc) \\
3 & : & \texttt{jz}\ l\ 14 \\
4 & : & \texttt{jmp}\ S0 \\
5 = Send & : & \texttt{tag}\ l_2 \\
6 & : & l := copy(l_1) \\
7 & : & \texttt{call}\ 0 \\
8 & : & l_1 := copy(l') \\
9 & : & \texttt{tag}\ l_1\ ;\ \texttt{untag}\ l_2 \\
10 & : & l := copy(l_2) \\
11 & : & \texttt{call}\ 0 \\
12 & : & l_2 := copy(l') \\
13 & : & \texttt{untag}\ l_1 \\
14 & : & a = \texttt{hd}(acc) \\
15 & : & \texttt{jmp}\ C0 \\
16 & : & \texttt{return} \\
end & &
\end{array}$$

Si on trace le réseau, on peut voir qu'il termine toujours à condition de prendre en compte l'unicité du compteur ordinal. On peut aussi voir que le réseau est sans allocation de mémoire, donc le programme est NSI.

Si on regarde aussi la consommation de mémoire dans les piles (et pas uniquement dans les variables, donc sur le tas), on peut voir que le réseau a besoin d'une quantité non bornée de mémoire à cet endroit et que le programme calcule donc en place vis-à-vis du tas mais pas vis-à-vis de la pile.

Chapitre 9

Analyse des réseaux de Petri

Ce chapitre présente des méthodes algébriques pour analyser les réseaux de Petri, et plus particulièrement pour détecter la terminaison d'un SCPN sous les conditions d'unicité du compteur ordinal et de cohérence (vis-à-vis des variables) de l'exécution infinie cherchée.

9.1 Notations

Définition 101 (Notation des vecteurs et des matrices).
Dans tout ce chapitre, quand un vecteur est utilisé, il peut être représenté indifféremment par un vecteur-colonne ou un vecteur-ligne, le contexte (multiplication matricielle en général) permettant de déterminer quelle est la bonne représentation. Ainsi, les transpositions des vecteurs ne sont pas notées.

Si aucune précision supplémentaire n'est faite, tous les vecteurs et toutes les matrices sont à coordonnées entières (positives ou négatives).

Si V est un vecteur, $V(i)$ représente la i-ème composante de V. De même, si M est une matrice, $M(i,j)$ représente la valeur de la case (i,j) de M.

Si E est un ensemble fini, on se donne une énumération arbitraire $x_1, \ldots, x_n$ des éléments de E. Si V est un vecteur à $n = |E|$ composantes, on peut noter $V(x_i)$ la i-ème composante de V. C'est-à-dire que le vecteur est identifié avec une fonction de E vers $\mathbb{Z}$. Cette notation est étendue aux matrices.

Définition 102 (Vecteur nul, vecteur unité).
$\mathbf{0}$ (resp. $\mathbf{1}$) représente le vecteur ou la matrice dont toutes les composantes valent 0 (resp. 1).

Définition 103 (Ordre sur les vecteurs).
Si A et B sont deux vecteurs de même dimension, on note :
- $A \geq B$ si pour tout i, $A(i) \geq B(i)$.
- $A \gneq B$ si $A \geq B$ et $A \neq B$.
- $A > B$ si pour tout i, $A(i) > B(i)$.

Définition 104 (Support d'un vecteur, support d'un ensemble).
Soit V un vecteur à composantes entières positives ou nulles. Son *support* $V_{\mathbf{1}}$ est le vecteur obtenu en remplaçant chaque composante non nulle par 1. C'est-à-dire que $V_{\mathbf{1}}(x) = min(V(x), 1)$.

Soient E un ensemble fini et $E' \subset E$ un sous-ensemble de E. Le *support* de E' est un vecteur $E'_{\mathbf{1}}$ à $|E|$ composantes tel que $E'_{\mathbf{1}}(x) = 1$ si $x \in E'$ et 0 sinon.

Réciproquement, si E est un ensemble fini et V un vecteur à $|E|$ composantes, on considère $V_{\mathbf{1}}$ comme le sous-ensemble de E qui contient tous les x tels que $V_{\mathbf{1}}(x) \neq 0$.

Lemme 56.
Soit V un vecteur à composantes positives ou nulles. Son support $V_{\mathbf{1}}$ est la solution du système :

$X \leq V$
$\mathbf{0} \leq X \leq \mathbf{1}$
$X \times V = \mathbf{1} \times V$

Démonstration.
V_1 est clairement solution du système.

Si X est solution du système, la deuxième équation implique que les composantes de X valent 0 ou 1. La première implique qu'une composante de X vaut 0 quand la composante correspondante de V vaut 0. La troisième implique qu'une composante de X vaut 1 quand la composante correspondante de V vaut 1 (sinon la somme serait plus petite). □

Définition 105 (Matrice diagonale).
Soit V un vecteur. La *matrice diagonale* $\overline{V}$ associée à V est la matrice carrée dont la diagonale est V et tous les autres coefficients valent 0
C'est-à-dire que $\overline{V}(i,i) = V(i)$ et $\overline{V}(i,j) = 0$ si $i \neq j$.

Lemme 57.
Soit V un vecteur. Sa matrice diagonale associée est la solution de l'équation :

$$M = \sum_i a_i V a_i b_i$$

*où les a_i sont les vecteurs-colonne de la base canonique (*ie *un 1 en i-ème position et des 0 ailleurs), les b_i sont les vecteurs lignes de la base canonique et V est considéré comme un vecteur ligne.*

Démonstration.
Le produit $a_i b_i$ est une matrice qui comporte un 1 sur la i-ème case de la diagonale et des 0 ailleurs. Le produit $a_i V$ est le scalaire $V(i)$ (la i-ème composante de V). Chaque élément de la somme est donc une matrice dont la i-ème case de la diagonale vaut $V(i)$ et tout le reste vaut 0. La somme est bien égale à la matrice diagonale associée à V. □

Définition 106.
Soit V un vecteur tel que $\mathbf{0} \leq V \leq \mathbf{1}$. On note $\Lambda(V)$ l'ensemble des vecteurs V' tels qu'il existe i qui vérifie :

- $V(i) = 1$
- $V'(j) = 0$ si $V(j) = 0$
- $V'(i) = 1 - \sum_j V(j)$
- $V'(k) = 1$ pour tout $k \neq i$ tel que $V(k) = 1$

Lemme 58.
Soit V un vecteur tel que $\mathbf{0} \leq V \leq \mathbf{1}$. $\Lambda(V)$ est l'ensemble des solutions X du système :

$Y \cdot \mathbf{1} = 1$
$\mathbf{0} \lneq Y \lneq V$
$X = V - (V \cdot \mathbf{1}) \cdot Y$

Démonstration.
Y est un vecteur composé uniquement de 0, sauf une composante qui vaut 1 et qui correspond à une composante non nulle de V. Donc X décrit bien l'ensemble des solutions. □

Définition 107.
Soit Γ une matrice composée de -1, 0 et 1. On note Γ^+ (resp. Γ^-) la matrice dont chaque composante vaut $\Gamma^+(i,j) = \max(0, \Gamma(i,j))$ (resp. $\Gamma^-(i,j) = \min(0, \Gamma(i,j))$).

9.2 Terminaison d'un réseau de Petri

Théorème 59 (Y. E. Lien 1976, **[Lie76]).**
Soient N un réseau de Petri et Γ sa matrice caractéristique. Décider si N est fortement non-terminant est dans NPTIME. *Ce qui revient à décider si il existe une solution entière au système d'inéquations suivant :*

$$\begin{aligned} \Gamma \times X &\geq \mathbf{0} \\ X &\geq \mathbf{1} \end{aligned}$$

Démonstration.
Si N est fortement non-terminant, il existe un marquage M_0 qui active une exécution ω dans laquelle chaque transition apparaît une infinité de fois. Ce marquage et cette exécution définissent une suite de marquages $(M)_n$ (chacun étant obtenu à partir du précédent en tirant une des transitions de ω dans l'ordre où elles apparaissent). De cette suite, on extrait une suite $(M')_n$ telle que chaque transition soit tirée au moins une fois entre M'_i et M'_{i+1}.

Comme l'ordre $\trianglelefteq$ est bien fondé, il existe deux marquages M'_i et M'_j tels que $i < j$ et $M'_i \trianglelefteq M'_j$. Par construction, il existe une séquence de tirs ω' telle que $M'_i \xrightarrow{\omega'} M'_j$ et chaque transition apparaît au moins une fois dans ω'.

Algébriquement, en posant $X = F_{\omega'}$, on obtient $M'_i + \Gamma \times X = M'_j$ (séquence de tirs), $X \geq \mathbf{1}$ (chaque transition apparaît au moins une fois) et $M'_i \leq M'_j$ (ordre sur les marquages). En reportant la troisième inéquation dans la première, on obtient $M'_i + \Gamma \times X \geq M'_i$, donc $\Gamma \times X \geq \mathbf{0}$.

Réciproquement, si le système d'inéquations admet une solution, on peut facilement construire une séquence de tirs ω telle que $F_\omega = X$ et un marquage M qui l'active. D'après la première inéquation, $M \xrightarrow{\omega} M' \trianglerighteq M$, donc M' active ω et par récurrence, ω peut être tirée une infinité de fois, conduisant à une exécution infinie. D'après la deuxième inéquation, chaque transition est tirée au moins une fois par ω, donc l'exécution infinie construite précédemment tire chaque transition une infinité de fois, le réseau est fortement non-terminant. □

Théorème 60.
Décider si un réseau de Petri est non-terminant est dans PTIME. *Ce qui revient à savoir si il existe une solution entière au système d'inéquations :*

$$\begin{array}{rcl} \Gamma \times X & \geq & \mathbf{0} \\ X & \gneq & \mathbf{0} \end{array}$$

Démonstration.
La preuve est similaire à celle du théorème précédent. La deuxième inéquation indique cette fois que les transitions ne peuvent pas être tirées un nombre négatif de fois, mais n'oblige pas à tirer chaque transition au moins une fois (donc dans l'exécution obtenue en répétant la séquence une infinité de fois, il peut y avoir des transitions qui n'apparaissent pas). La deuxième inéquation indique aussi qu'au moins une transition doit être tirée dans la "boucle" qui est construite et permet d'éliminer le cas trivial $X = 0$.

Clairement, si X est une solution du système, alors $c \cdot X$ (où c est une constante quelconque) est aussi une solution. L'existence de solutions dans $\mathbb{N}$ est donc équivalente à l'existence de solutions dans $\mathbb{Q}$. De plus, la forme du système permet de voir que l'ensemble des solutions forme un cône de l'espace. En conséquence, si il existe une solution dans $\mathbb{R}$, il existe aussi une solution dans $\mathbb{Q}$. Comme seule l'existence d'une solution (ou plutôt sa non-existence) est pertinente pour le moment (la valeur exacte de la solution n'a aucune importance), il suffit de chercher si il existe une solution dans $\mathbb{R}$. Or ce dernier problème (résolution de système d'inéquations linéaires dans $\mathbb{R}$) est connu pour être dans PTIME. □

Exemple 23.

1. La matrice caractéristique du SCPN de l'addition (version récursion terminale) est la suivante :

	$x = 0$	$x \neq 0$	`dec` x	`inc`y	`jmp` 0
0	-1	-1	0	0	1
1	0	1	-1	0	0
2	0	0	1	-1	0
3	0	0	0	1	-1
end	1	0	0	0	0
x	0	0	-1	0	0
y	0	0	0	1	0

La seule solution du système $\Gamma \cdot X \geq \mathbf{0}$ est $X = \mathbf{0}$. Le SCPN n'a donc pas d'exécution infinie, donc le programme termine toujours.

La matrice caractéristique du SCPN de l'addition (deuxième version) est la suivante (les 0 ont été remplacés par des cases vides) :

	`call` 1 (1)	$x=0$	$x \neq 0$	`dec` x	`call` 1 (3)	ret_3	`inc` y	`return`	ret_0
0	-1								
1	1	-1	-1		1				
2			1	-1					
3				1	-1				
4						1	-1		
5		1					1	-1	
end									1
`return`$_1$						-1		1	-1
$stack_0$	1								-1
$stack_3$					1	-1			
x				-1					
y							1		

Là encore, la seule solution du système $\Gamma \cdot X \geq \mathbf{0}$ est $X = \mathbf{0}$ donc le SCPN termine, donc la machine à compteurs n'a pas d'exécution infinie.

2. La matrice caractéristique de la macro $x := copy(y)$ est :

	$y=0$	$y \neq 0$	`dec` y	`inc`$_x$	`jmp` 0
0	-1	-1			1
1		1	-1		
2			1	-1	
3				1	-1
end	1				
x				1	
y			-1		

La seule solution du système $\Gamma \cdot X \geq \mathbf{0}$ étant $X = \mathbf{0}$, le réseau de Petri termine donc la machine termine toujours.

9.3 Unicité du compteur ordinal

Exemple 24.
Pour la macro $x := dup(y)$, on a la matrice suivante (les deux "boucles" sont représentées ici par deux transitions chacune) :

	$y \neq 0$	`inc` y'	$y=0$	$y' \neq 0$	`inc` y	$y'=0$
0	-1	1	-1			
2	1	-1				
5			1	-1	1	-1
7				1	-1	
end						1
x	1					
y	-1		1			
y'	1		-1			

Là, le système $\Gamma \cdot X \geq \mathbf{0}$ admet des solutions strictement positives, de la forme $(k, k, 0, k, k, 0)$ où k est un entier. En effet, les séquences de tirs correspondantes peuvent être tirées en boucle et le réseau ne termine pas. Par contre, on peut voir que ces séquences de tirs ne sont activées que si il y a (au moins) un jeton dans 0 *et* un dans 5 ce qui ne correspond à aucun état légal du système (car il y aurait alors deux jetons ordinaux). Donc le réseau de Petri termine *pour tout état initial*, donc le programme termine toujours.

Ce problème se ramène facilement à celui du marquage initial minimum qui consiste, étant donné un vecteur X à trouver un marquage minimum qui active une séquence de tirs ω telle que $F_\omega = X$.

Ce problème a été montré NPTIME-difficile par T. Watanabe, Y. Mizobata et K. Onaga [WMO89]. Malheureusement, la méthode qu'ils utilisent pour le résoudre n'est pas applicable dans notre cas. En effet, elle repose sur le fait qu'ils ne testent qu'une seule séquence de tirs (un seul vecteur X) alors qu'ici il y a une infinité de vecteurs X (toutes les solutions du système).

Heureusement, on est ici dans un cas particulier qui permet d'envisager une autre technique. En effet, la forme du réseau de Petri permet de se concentrer uniquement sur le CFPN vu comme un graphe et à ramener le problème d'unicité du compteur ordinal à un problème de connexité, qui est lui-même ramené à un calcul de flot.

Je commence par présenter la méthode pour détecter si un graphe orienté est connexe.

9.3.1 Connexité d'un graphe, calcul de flot

Définition 108 (Connexité, connexité forte).
Soit $G = (V, E)$ un graphe orienté. G est *connexe* si il existe un sommet s tel que pour tout sommet $u \neq s$ il existe un chemin qui va de s à u. G est *fortement connexe* si pour tout couple de sommets u, v il existe un chemin qui va de u à v.

Remarque :
Cette définition de la connexité n'est pas la définition usuelle. En effet, un graphe avec trois sommets a, b, c et deux arcs $(a, b), (c, b)$ est connexe au sens usuel du terme mais pas au sens défini ci-dessus.

La matrice d'incidence d'un graphe orienté est une matrice qui a autant de lignes que le graphe a de sommets et autant de colonnes que le graphe a d'arêtes. Les coefficients valent -1 si l'arête part du sommet, 1 si elle y arrive et 0 sinon.

Définition 109 (Matrice d'incidence).
Soit $G = (V, E)$ un graphe orienté. Sa *matrice d'incidence* I_G est une matrice $|V| \times |E|$ avec

- $I_G(u, (u, v)) = -1$
- $I_G(v, (u, v)) = 1$
- $I_G(a, (u, v)) = 0$ si a n'est ni u ni v.

Définition 110 (flot).
Soit $G = (V, E)$ un graphe orienté. Un flot f est une fonction qui assigne à chaque sommet et à chaque arc une valeur telle que pour tout sommet u, $f(u) = \sum_{v/(u,v)\in E} f(u, v) - \sum_{v/(v,u)\in E} = f(v, u)$.

C'est-à-dire que le flot de chaque sommet est égal à la différence entre les sommes des flots sur les arcs entrants et des flots sur les arcs sortants.

Un flot peut être représenté comme deux vecteurs, l'un a $|V|$ composantes indique la valeur du flot pour chaque sommet et l'autre à $|E|$ composantes indique la valeur du flot sur chaque arc.

Lemme 61.
Soient $G = (V, E)$ un graphe et X un vecteur à $|E|$ composantes. Il existe un flot f_X défini par $f_X(u, v) = X(u, v)$ et $f_X(u) = (I_G \times X)(u)$.

Démonstration.
f_X vérifie bien les conditions de la définition précédente. □

Lemme 62.
Soient G un graphe orienté, I_G sa matrice d'incidence, s et t deux sommets. Il existe un chemin de s vers t si et seulement si il existe un flot f tel que $f(s) = -1$ et $f(t) = 1$ et $f(u) = 0$ pour tous les autres sommets.

Démonstration.
Si le chemin existe, on prend $f(u, v) = 1$ sur chaque arc du chemin et 0 ailleurs.
Si le flot existe, on peut construire un chemin (pas forcément minimal) qui passe $f(u, v)$ fois sur chaque arc (u, v). □

Corollaire 63.
Soient $G = (V, E)$ un graphe orienté, I_G sa matrice d'incidence, s et t deux sommets. Soit A le vecteur

à $|V|$ composantes tel que $A(s) = -1$, $A(t) = 1$ et $A(u) = 0$ pour les autres sommets.

Il existe un chemin de s vers t si et seulement si le système d'équations $I_G \times X = A$ admet une solution.

Démonstration.
C'est une conséquence directe des deux lemmes précédents. X et A sont les deux vecteurs décrivant le flot. □

Théorème 64.
Soient $G = (V, E)$ un graphe orienté, I_G sa matrice d'incidence et s un sommet. Soit A^s le vecteur à $|V|$ composantes tel que $A^s(s) = 1 - |V|$ et $A^s(u) = 1$ pour chaque autre sommet.

Tous les sommets sont accessibles depuis s si et seulement si le système $I_G \times X = A^s$ admet une solution.

Démonstration.
Cela revient à dire qu'il existe un flot unitaire depuis s vers chacun des autres sommets du graphe et à combiner ces flots en un seul. □

Corollaire 65.
Soient G un graphe orienté et I_G sa matrice d'incidence.
G est connexe si et seulement si il existe un sommet s tel que le système $I_G \times X = A^s$ admet une solution.
G est fortement connexe si et seulement si pour tout sommet u, le système $I_G \times X = A^u$ admet une solution.

Théorème 66.
Soit G un graphe orienté tel que chaque arête appartient à au moins un cycle. G est fortement connexe si et seulement si il est connexe.

Démonstration.
Si G est connexe, il existe un sommet s tel que pour tout sommet u il existe un chemin de s vers u. Par récurrence sur la longueur de ce chemin, on peut construire un chemin de u vers s.

En effet, on considère v le sommet atteint juste avant u sur ce chemin. Par hypothèse, l'arc (v, u) fait partie d'un cycle. Il existe donc un chemin de u vers v. Par hypothèse de récurrence, il existe un chemin de v vers s.

Si on considère maintenant deux sommets quelconques u et v, il existe un chemin de u vers s et un chemin de s vers v. Donc il existe un chemin de u vers v, le graphe est fortement connexe.

La réciproque s'obtient parce que la forte connexité implique toujours la connexité. □

9.3.2 Sous-graphes

On construit le sous-graphe à partir d'un ensemble d'arêtes en ne conservant que ces arêtes et les sommets qui sont à leurs extrémités.

Définition 111 (Sous-graphe).
Soient $G = (V, E)$ un graphe, I_G sa matrice d'incidence et $E' \subset E$ un ensemble d'arêtes. Le *sous-graphe* $G_{|E'}$ extrait de G à partir de E' est tel que :
- L'ensemble des sommets de $G_{|E'}$ est $\{u \in V/(u, v) \in E' \text{ ou } (v, u) \in E'\}$.
- L'ensemble des arcs de $G_{|E'}$ est E'.

Lemme 67.
Soient $G = (V, E)$ un graphe orienté, I_G sa matrice d'incidence et $E' \subset E$ un ensemble d'arcs. La matrice d'incidence du sous-graphe extrait à partir de E' est le produit de la matrice d'incidence du graphe et de la matrice diagonale associée au support de E'.

$$I_{G_{|E'}} = I_G \times \overline{E'_{\mathbf{1}}}$$

En fait, la matrice obtenue est plus importante que la matrice d'incidence du sous-graphe : elle comporte en plus des lignes et des colonnes de 0 correspondant aux sommets et aux arêtes qui ont été supprimés.

Démonstration.
La multiplication par $\overline{E'_1}$ annule toutes les colonnes qui ne correspondent pas à des arcs de E'. Les lignes qui correspondent à des sommets qui ne sont pas dans $G_{|E'}$ sont annulées car leurs seules valeurs non nulles dans I_G sont celles qui correspondent à des arcs supprimés. □

9.3.3 CFPN et graphe sous-jacent

Définition 112 (Graphe de places).
Soit N un réseau de Petri tel que chaque transition a exactement une place d'entrée et une place de sortie. Son *graphe de places* G est le graphe orienté qui a un sommet par place de N et un arc par transition.

La matrice caractéristique du réseau de Petri est égale à la matrice d'incidence de son graphe de places : $\Gamma_N = I_G$.

Les places de N sont confondues avec les sommets de son graphe de places et les transitions sont confondues avec les arcs du graphe de places.

Lemme 68.
Soient N un réseau et G son graphe de places. Soit ω une séquence de tirs de N' qui est activée par un marquage avec un seul jeton. ω est un chemin dans G.

Démonstration.
Par construction du graphe de places. □

Définition 113.
Soient `pgm` une machine à compteurs, N son SCPN et N' son CFPN, Γ la matrice caractéristique de N.

On note Γ' la matrice caractéristique de N' qui est obtenue à partir de Γ en supprimant les lignes correspondant aux places qui ne sont pas ordinales. Comme le CFPN correspond aux conditions de la définition précédente, Γ' est aussi la matrice d'incidence du graphe de places de N'.

Lemme 69.
Soient `pgm` une machine à compteur, N son SCPN, N' son CFPN, $M_i \trianglelefteq M_f$ deux marquages de N et ω une séquence de tirs telle que $M_i \xrightarrow{\omega} M_f$.

Soient M'_i et M'_f les marquages de N' qui coïncident avec M_i et M_f sur les places ordinales. On a $M'_i = M'_f$

Démonstration.
On a $M'_i \trianglelefteq M'_f$ par construction des deux marquages. Comme le nombre de jetons dans les places ordinales est constant, et que M'_i et M'_f ne concernent que les places ordinales, on a $M'_i = M'_f$. □

Corollaire 70.
Soient Γ et Γ' les matrices caractéristiques de N et N' et X un vecteur.

Si $\Gamma \times X \geq \mathbf{0}$ alors $\Gamma' \times X = \mathbf{0}$.

Démonstration.
Par application des équations qui régissent le comportement des réseaux de Petri. □

Corollaire 71.
Soient N' un CFPN et Γ' sa matrice caractéristique. Soit G le graphe de places de N'. Soit V un vecteur tel que $\Gamma' \times V = \mathbf{0}$. $G_{|V_1}$ est composé uniquement de cycles.

Démonstration.
Car chaque jeton dans une place de N' se retrouve dans la même place après avoir tiré toutes les transitions de V. Donc il y a un chemin depuis chaque sommet de $G_{|V_1}$ vers lui-même. □

Théorème 72.
Soient N' un CFPN, ω une séquence de tirs qui laisse N' invariant (c'est-à-dire que $M \xrightarrow{\omega} M$ pour tout marquage qui active ω) et $F = F_\omega$. Soit G le graphe de places de N'.

$G' = G_{|F_1}$ est fortement connexe si et seulement si il existe une séquence de tirs ω' activée par un marquage avec un seul jeton et telle que $F_{\omega'} = F_\omega$.

Démonstration.
Si G' n'est pas fortement connexe, comme G' est composé uniquement de cycles (d'après le lemme précédent), G' n'est pas connexe. Donc il n'existe pas de chemin dans G' qui passe par tous les arcs. Donc il n'existe pas de séquence de tir activable avec un seul jeton qui tire toutes les transitions correspondant aux arcs de G'. Comme ω tire toutes les transitions correspondant aux arcs de G', ω n'est pas activée par un marquage avec un seul jeton.

Réciproquement, si G' est fortement connexe, on raisonne par récurrence sur le nombre de jetons nécessaire à activer la séquence de tirs. Considérons une séquence de tirs ω' telle que $F_{\omega'} = F_\omega$ qui soit activée par un marquage M_0 avec deux jetons. Soient p et p' deux places de N' telles que $M_0(p)+M_0(p') > 1$ (on peut avoir $p = p'$).

Comme G' est fortement connexe, p' est accessible depuis p et réciproquement. On peut donc construire à partir de ω' une séquence de tirs qui est activée par un marquage M'_0 avec un seul jeton. En effet, on prend $M'_0(p) = 1$, on tire toutes les transitions nécessaires à obtenir un marquage M'_1 avec $M'_1(p') = 1$, puis toutes les transitions qui constituent la “boucle” qui partait de p', puis les transition qui restent et qui permettent de revenir à M'_0. □

9.3.4 Mise en équations

On arrive ainsi en utilisant les propriétés du CFPN à ramener le problème d'unicité du compteur ordinal à un problème de connexité dans un graphe. Il ne reste plus qu'à rassembler tous les morceaux pour mettre cette condition sous forme de système d'inéquations linéaires.

Lemme 73.
Soient $G = (V, E)$ un graphe et $E' \subset E$ un ensemble de cycles (c'est-à-dire un ensemble d'arcs faisant tous partie d'un cycle). Soit $G' = G_{|E'} = (V', E')$ le graphe extrait. Soient I et I' les matrices d'incidences.

Soit A un vecteur à $|V|$ composantes tel que il existe un $u \in V'$ qui vérifie :
- *$A(u) = 1 - |V'|$*
- *$A(v) = 1$ si $v \in V'$ et $v \neq u$*
- *$A(v) = 0$ sinon*

(c'est-à-dire $A \in \Lambda(V'_{\mathbf{1}})$).

G' est fortement connexe si et seulement si le système $I' \times X = A$ a une solution. De plus,

$$V'_{\mathbf{1}} = I'^{+} \cdot \mathbf{1} = I^{+} \cdot E'_{\mathbf{1}}$$

Démonstration.
X et A décrivent un flot qui relie le sommet u à tous les autres et comme G' est composé uniquement de cycles, la connexité implique la forte connexité.

La première égalité matricielle est obtenue car $\underline{I'^{+}}$ ne contient des 1 que dans les lignes correspondant à $V'_{\mathbf{1}}$. La deuxième découle du fait que $I' \cdot \mathbf{1} = I \cdot \overline{E'_{\mathbf{1}}} \cdot \mathbf{1} = I \cdot E'_{\mathbf{1}}$ (lemme 67). □

Corollaire 74.
Soient N un CFPN, G son graphe de places, $\Gamma = I$ la matrice caractéristique de N (et matrice d'adjacence de G).

Soient $A \gneq \mathbf{0}$ un vecteur tel que $\Gamma \times A \geq \mathbf{0}$ et $G_{|A_{\mathbf{1}}} = (V', E')$. Les propositions suivantes sont équivalentes :

1. *Il existe une séquence de tirs ω dans N telle que $F_\omega = A$ et ω est activée par un marquage de N avec un seul jeton.*
2. *$G_{|A_{\mathbf{1}}}$ est connexe.*
3. *$G_{|A_{\mathbf{1}}}$ est fortement connexe.*
4. *Le système $\Gamma' \cdot X = A'$ admet une solution, avec $\Gamma' = \Gamma \cdot \overline{A_{\mathbf{1}}}$ la matrice d'incidence de $G_{|A_{\mathbf{1}}}$ et $A' \in \Lambda(V'_{\mathbf{1}})$.*

Démonstration.
2 et 3 sont équivalent car $G_{A_{\mathbf{1}}}$ est composé uniquement de cycles (théorème 71).
1 et 3 sont équivalents d'après le théorème 72.
3 et 4 sont équivalents d'après le lemme précédent. □

Théorème 75.
Soient $\texttt{pgm}$ *une machine à compteurs,* N *son SCPN,* N' *son CFPN,* Γ *et* Γ' *les matrices caractéristiques.* N *admet une exécution infinie qui ne nécessite qu'un seul jeton ordinal si le système linéaire suivant admet une solution :*

$$\begin{array}{l} \Gamma \cdot X \geq \mathbf{0} \\ X \gneq \mathbf{0} \\ V \in \Lambda(\Gamma'^{+} \cdot X_{\mathbf{1}}) \\ \Gamma' \cdot \overline{X_{\mathbf{1}}} \cdot Y = V \end{array}$$

Démonstration.
Les deux premières inéquations calculent X qui est le vecteur caractéristique d'une séquence de tirs de N qui peut être tirée infiniment.

Les deux dernières vérifient qu'il existe une séquence de tir activable avec un seul jeton ordinal dont le vecteur caractéristique est X. □

Γ , Γ' et Γ'^{+} ne dépendent que du réseau. $X_{\mathbf{1}}$, $\overline{X_{\mathbf{1}}}$ et $\Lambda(V')$ sont des solutions de systèmes d'inéquations linéaires. On peut donc regrouper l'ensemble de ces conditions en un seul système d'inéquations linéaires :

(calcul de la séquence de tirs qui boucle)

$$\begin{array}{l} \Gamma \cdot X \geq \mathbf{0} \\ X \gneq \mathbf{0} \end{array}$$

(calcul du support de cette séquence)

$$\begin{array}{l} X_{\mathbf{1}} \leq X \\ \mathbf{0} \leq X_{\mathbf{1}} \leq \mathbf{1} \\ X_{\mathbf{1}} \times X = \mathbf{1} \times X \end{array}$$

(calcul d'un flot sur les places ordinales atteintes)

$$\begin{array}{l} X' = \Gamma'^{+} \cdot X_{\mathbf{1}} \\ Y \cdot \mathbf{1} = 1 \\ \mathbf{0} \lneq Y \lneq V \\ V = X' - (X' \cdot \mathbf{1}) \cdot Y \end{array}$$

(calcul de la matrice diagonale associée à $X_{\mathbf{1}}$)

$$\overline{X_{\mathbf{1}}} = \sum_i a_i V a_i b_i$$

(connexité du sous-graphe)

$$\Gamma' \cdot \overline{X_{\mathbf{1}}} \cdot Z = V$$

Les inconnues sont X, Y et Z et si une solution existe, X est le vecteur caractéristique de la séquence de tirs correspondante.

La taille du système est polynomiale dans la taille du réseau, donc dans la taille de la machine à compteurs.

Savoir si il existe une solution entière est dans NPTIME. En effet, il suffit de "deviner" la valeur de la solution.

Exemple 25.
Si on reprend l'exemple du *dup*, les solutions de $\Gamma \cdot X \geq 0$ sont de la forme $X = (k, k, 0, k, k, 0)$. On a donc $X_{\mathbf{1}} = (1, 1, 0, 1, 1, 0)$. On trouve donc $X' = (1, 1, 1, 1, 0)$, ce qui donne quatre possibilités pour V : $V_1 = (-3, 1, 1, 1, 0)$, $V_2 = (1, -3, 1, 1, 0)$, $V_3 = (1, 1, -3, 1, 0)$ et $V_4 = (1, 1, 1, -3, 0)$. Mais comme la colonne $y = 0$ de $\Gamma' \cdot \overline{X_{\mathbf{1}}}$ est remplie de 0, la dernière inéquation n'a pas de solution quel que soit le V choisi parmi les quatre possibles.

Donc la macro *dup* termine toujours.

9.4 Cohérence vis-à-vis des variables

Exemple 26.
La matrice caractéristique du SCPN de l'algorithme d'Euclide est la suivante (les sept "boîtes" ont été nommées de A à G dans le sens des aiguilles d'une montre, les trois transitions $b = 0$ n'ont pas été

renommées) :

	$b=0$	$b\neq 0$	$a<b$	$a\geq b$	$a=0$	$b=0$	A	B	C	D	E	F	G	$b=0$
0	-1	-1					1						1	
1		1	-1	-1										
2				1										-1
7													-1	1
9			1		-1									
14					1	-1								
15						1	-1							
end	1													
a									1	-1	-1			
b								1	-1	-1	-1	1		
tmp								-1		1	1	-1		

Le système $\Gamma \cdot X \geq 0$ admet des solutions non nulles, même à partir d'un état initial. En effet, on a (par exemple) l'exécution infinie du réseau de Petri : $b \neq 0, a \geq b, b = 0, G$. Le problème vient cette fois-ci de la cohérence vis-à-vis de la variable b, c'est-à-dire qu'on ne peut pas avoir $b \neq 0$ puis $b = 0$ sans que la valeur de b ait été modifiée entre temps.

Comme on l'a vu précédemment, ce problème peut se résoudre en augmentant la taille du réseau de Petri pour éliminer directement les exécutions non cohérentes. Cependant, il existe aussi une version algébrique de ces contraintes.

Théorème 76.
Soient N un SCPN et Γ sa matrice caractéristique. N admet une exécution infinie cohérente vis-à-vis des variables si il existe un $X \gneq \mathbf{0}$ tel que $\Gamma \times X \geq \mathbf{0}$ et pour chaque variable x,

$$((X(x=0)>0) \wedge (X(x\neq 0)>0)) \Rightarrow ((X(\texttt{inc}\ x)>0) \wedge (X(\texttt{dec}\ x)>0))$$

Si il existe plusieurs transitions correspondant à des instructions identiques (*eg* plusieurs $b = 0$ pour l'algorithme d'Euclide), $X(b = 0)$ représente ici la somme des composantes de X correspondant à chacune de ces transitions.

Démonstration.
La condition supplémentaire élimine les exécutions où $x = 0$ et $x \neq 0$ sont tirées sans que la valeur x soit augmentée ou diminuée entre temps. □

Cette condition logique supplémentaire peut aussi se mettre sous forme de système linéaire. Pour cela, il faut prendre une variable booléenne pour chacune des inégalités dans l'implication (donc ici quatre variables booléennes) et mettre la condition en forme normale conjonctive. Chaque disjonction de la forme normale conjonctive correspond alors à une inéquation supplémentaire.

Théorème 77.
Soient x, y, z, t quatre entiers naturels. L'implication $((x > 0) \wedge (y > 0)) \Rightarrow ((z > 0) \wedge (t > 0))$ est vraie si et seulement si le système suivant admet une solution (entière) :

$$\begin{array}{l}
\mathbf{0} \leq (a,b,c,d) \leq \mathbf{1} \\
(1-a)+(1-b)+c+d>0 \\
(1-a)+(1-b)+c+(1-d)>0 \\
(1-a)+(1-b)+(1-c)+d>0 \\
(c,d) \leq (z,t) \\
a \cdot x = x \\
b \cdot y = y
\end{array}$$

Démonstration.
Si $x = 0$, l'implication est vraie et en prenant $a = 0$, le système a une solution.
Idem si $y = 0$ en prenant $b = 0$.
Si x, y, z, t sont tous les quatre non-nuls, l'implication est vraie et le système admet une solution en prenant $a = b = c = d = 1$.

Si x et y sont non nuls et que z est nul, l'implication est fausse. x et y non nuls forcent $a = b = 1$, z nul force $c = 0$. Si $d = 0$ on a $(1-a)+(1-b)+c+d = 0$ et si $d = 1$ on a $(1-a)+(1-b)+c+(1-d) = 0$, donc le système n'admet pas de solution.
De même si x et y sont non nuls et $d = 0$. $\square$

Donc on peut forcer la cohérence de l'exécution tirée en ajoutant autant de copies de ce système que de variables dans la machine à compteurs. Là encore, le système reste de taille polynomiale par rapport à la taille du réseau et savoir si il existe une solution au système est dans NPTIME.

On peut ainsi prouver la terminaison de l'algorithme d'Euclide.

Bien sûr, il est possible de combiner les contraintes de cohérence avec les contraintes d'unicité du compteur ordinal en un seul gros système d'inéquations, dont la taille reste polynomiale dans la taille du réseau.

Conclusions et perspectives

Cette thèse a présenté deux résultats principaux. Le premier caractérise des classes de complexité usuelles (PTIME et PSPACE) au moyen de restrictions syntaxiques sur les programmes (ordres de terminaison et quasi-interprétations). Ce résultat a deux intérêts principaux. D'une part il s'agit bel et bien d'une caractérisation de la complexité *implicite* de l'algorithme et pas de sa complexité explicite. La transformation effectuée permet en outre d'obtenir des accélérations exponentielles entre le programme originel (sans mémoïsation) et le programme final (avec mémoïsation). On peut ainsi faire automatiquement de la programmation dynamique quand cela s'avère nécessaire. D'autre part, cette caractérisation capture beaucoup plus de "bons" algorithmes que les précédentes, sa complétude intentionnelle est donc plus grande. Il devient donc possible d'étudier la complexité d'une *spécification* et d'en déduire un algorithme efficace.

Mais ce résultat pourrait encore être étendu. Une première piste est, bien sûr, de chercher à capturer encore plus d'algorithmes (comme, par exemple, les algorithmes "Diviser pour régner"). Une autre piste est d'essayer de travailler avec des spécifications de niveau encore supérieur, par exemple une preuve obtenue dans un système de preuve formel comme Coq ou NuPrl. Ainsi, on pourrait obtenir un programme dont le comportement serait certifié par l'extraction de preuve et la complexité par l'analyse. Dans le même ordre d'idées, on pourrait imaginer d'utiliser les ordres de terminaison (couplés avec les quasi-interprétations) comme garde-fous lors d'une transformation de programme (par exemple une supercompilation). Ainsi, on pourrait obtenir un supercompilateur qui transforme le programme et en même temps donne une borne sur la complexité du programme final.

Le deuxième résultat analyse des propriétés extensionnelles des algorithmes au moyen de réseaux de Petri. Là aussi, cette analyse permet, à ma connaissance, de gérer plus de cas que les autres analyses existantes mais surtout de gérer plus de cas utiles en pratique (comme l'algorithme d'Euclide). La complétude intentionnelle est donc plus importante que celle des techniques précédentes.

Le gros problème de cette méthode vient de la faible expressivité du langage. Il est donc important d'essayer d'étendre le langage pour pouvoir écrire (et analyser) plus de programmes. Comme je l'ai montré dans les exemples, une extension pour gérer des listes ou des nombres binaires ne pose pas beaucoup de difficultés. Le défi qu'il faut relever est celui du stockage externe, que ce soit par le biais d'une pile permettant de conserver les valeurs des compteurs lors d'un `call` ou par le biais d'une "mémoire centrale" dans laquelle on pourrait lire et écrire des valeurs. On peut noter que l'un ou l'autre de ces ajouts permettrait au langage d'être vraiment très proche des langages assembleurs (ou bytecodes) réels. Il deviendrait alors possible d'analyser la complexité d'un code compilé (ou précompilé) et cette méthode pourrait devenir un composant à part entière des compilateurs.

Mais le principal intérêt de cette analyse vient du fait qu'elle permet de regrouper ensemble plusieurs propriétés des algorithmes et de les étudier simultanément. Ainsi, on peut à la fois savoir si l'algorithme termine, si il calcule en place et effectuer de la déforestation dessus. C'est là une piste très intéressante de recherche. Est-il possible de regrouper encore d'autres analyses ou techniques de transformation de programmes dans cette méthode ? Je pense que oui. Au cours de ma thèse, j'ai eu l'occasion de voir différentes méthodes d'analyse ou de transformation de programmes, et j'ai très souvent eu l'impression que ces méthodes se basaient toutes plus ou moins sur les mêmes deux ou trois techniques de base avec quelques variations mineures. Je pense qu'il est donc possible de regrouper toutes ces méthodes en un seul gros "outil universel" capable de factoriser les analyses en commun et de prouver rapidement une grande variété de propriétés tout en transformant le programme.

Ainsi, je pense par exemple qu'il serait possible d'adapter les ordres de terminaison et les quasi-interprétations aux réseaux de Petri. On pourrait donc, en même temps que les autres analyses, donner une borne sur la complexité du programme.

Cet “outil universel” de l’analyse et de la transformation de programmes n’est peut-être pas les réseaux de Petri, mais ils semblent être un bon candidat. Je pense donc qu’il est important de continuer à les étudier et à les comparer avec les techniques d’analyse et de transformation déjà existantes.

Bibliographie

[Ack28] Ackermann (W.), « Zum Hilbertschen Aufbau der reellen Zahlen », *Math. Annalen*, vol. 99, 1928, p. 118–133.

[AG00] Arts (T.) et Giesl (J.), « Termination of term rewriting using dependency pairs », *Theoretical Computer Science*, vol. 236, n° 1–2, 2000, p. 133–178.

[AJ94] Andersen (N.) et Jones (N. D.), « Generalizing Cook's transformation to imperative stack programs », dans Karhumäki (J.), Maurer (H.) et Rozenberg (G.), éditeurs, *Results and trends in theoretical computer science*, vol. 812 (coll. *Lecture Notes in Computer Science*), p. 1–18, 1994.

[Ama02] Amadio (R. M.), « Max-plus quasi-interprétations ». Rapport technique n° 10-2002, LIF, 2002. Extended abstract à apparaître dans TLCA 2003.

[AS00] Aehlig (K.) et Schwichtenberg (H.), « A syntactical analysis of Non-Size Increasing polynomial time computation », dans *Proceedings of the Fifteenth IEEE Symposium on Logic in Computer Science (LICS '00)*, p. 84–91, 2000.

[BAMD] Borgnat (P.), Asseray (B.), Moyen (J.-Y.) et Dubacq (J.-C.), « Europa Universalis à 8 joueurs ». Accessible depuis `http://bamgames.free.fr/Europa/`.

[BC92] Bellantoni (S. J.) et Cook (S. A.), « A new recursion-theoretic characterization of the polytime functions », *Computational Complexity*, vol. 2, 1992, p. 97–110.

[BCMT98] Bonfante (G.), Cichon (A.), Marion (J.-Y.) et Touzet (H.), « Complexity classes and rewrite systems with polynomial interpretation », dans *Computer Science Logic, 12th International Workshop, CSL'98*, vol. 1584 (coll. *Lecture Notes in Computer Science*), p. 372–384, 1998.

[BCMT00] Bonfante (G.), Cichon (A.), Marion (J.-Y.) et Touzet (H.), « Algorithms with polynomial interpretation termination proof », *Journal of Functional Programming*, vol. 11, 2000.

[Bel94] Bellantoni (S. J.), « Predicative recursion and the polytime hierarchy », dans Clote (P.) et Remmel (J.), éditeurs, *Feasible Mathematics II*, coll. « Perspectives in Computer Science ». Birkhäuser, 1994.

[BH00] Bellantoni (S. J.) et Hofmann (M.), « A new "feasible" arithmetic », *Journal of symbolic logic*, 2000. À apparaître.

[BMM01] Bonfante (G.), Marion (J.-Y.) et Moyen (J.-Y.), « On lexicographic termination ordering with space bound certifications », dans Bjørner (D.), Broy (M.) et Zamulin (A. V.), éditeurs, *PSI*, vol. 2244 (coll. *Lecture Notes in Computer Science*), p. 482–493. Springer, juillet 2001.

[BNS00] Bellantoni (S. J.), Niggl (K.-H.) et Schwichtenberg (H.), « Higher type recursion, ramification and polynomial time », *Annals of Pure and Applied Logic*, vol. 104, n° 1-3, 2000, p. 17–30.

[Chu38] Church (A.), « An unsolvable problem of elementary number theory », *American Journal of Mathematics*, n° 58, 1938, p. 345–363.

[Cic90] Cichon (A.), « Bounds on derivation lengths from termination proofs ». Rapport technique, University of London, Royal Holloway and Bedford New College, 1990.

[CKS81] Chandra (A.), Kozen (D.) et Stockmeyer (L.), « Alternation », *Journal of the ACM*, vol. 28, 1981, p. 114–133.

[CL87] Cherifa (A. B.) et Lescanne (P.), « Termination of rewriting systems by polynomial interpretations and its implementation », *Science of computer Programming*, 1987, p. 131–159.

[CLR94] Cormen (T. H.), Leiserson (C. E.) et Rivest (R. L.), *Introduction à l'algorithmique*. Dunod, 1994. Traduit de l'américain par X. Cazin.

[CM00] Cichon (A.) et Marion (J.-Y.), « The Light Lexicographic Path Ordering ». Rapport technique, Loria, 2000. Workshop Rule.

[Cob62] Cobham (A.), « The intrinsic computational difficulty of functions », dans Bar-Hillel (Y.), éditeur, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, p. 24–30. North-Holland, Amsterdam, 1962.

[Col98] Colson (L.), « Functions versus Algorithms », *EATCS Bulletin*, vol. 65, 1998. The logic in computer science column.

[Coo70] Cook (S. A.), « Path systems and language recognition », dans *Proceedings of the 2nd Annual ACM Symposium on the Theory of Computing*, p. 70–72, 1970.

[Coo71] Cook (S. A.), « Characterizations of pushdown machines in terms of time-bounded computers », *Journal of the ACM*, vol. 18, n° 1, janvier 1971, p. 4–18.

[Coo91] Cook (S. A.), « Computational Complexity of Higher Type Functions », dans *Proceedings of 1990 Intern. Congress of Mathematicians, Kyoto, Japon*, p. 55–69. Springer-Verlag, 1991.

[DE95] Desel (J.) et Esparza (J.), *Free Choice Petri Nets*. Cambridge University Press, 1995.

[D'E98] D'Eer (M.), *La bière – Ales, lagers et lambics*. Trécarré, 1998.

[Der82] Dershowitz (N.), « Orderings for term-rewriting systems », *Theoretical Computer Science*, vol. 17, n° 3, 1982, p. 279–301.

[Der87] Dershowitz (N.), « Termination of rewriting », *Journal of Symbolic Computation*, 1987, p. 69–115.

[DJ90] Dershowitz (N.) et Jouannaud (J.-P.), *Handbook of Theoretical Computer Science vol.B*, chap. Rewrite systems, p. 243–320. Elsevier Science Publishers B. V. (NorthHolland), 1990.

[DP02] Danner (N.) et Pollett (C.), « Minimization and NP multifunctions », *Theoretical Computer Science*, 2002. À apparaître.

[Esp98] Esparza (J.), « Decidability and Complexity of Petri Net Problems - An Introduction », *Lecture Notes in Computer Science*, vol. 1491, 1998, p. 374–428.

[Euc00] Euclide, *στοιχεία γεωμετρίας*, vol. VII. Alexandrie, environ -300. traduction anglaise : *The Thirteen Books of Euclid's Elements* par T. L. Heath, Dover Pubns, 1956.

[FGK02] Fissore (O.), Gnaedig (I.) et Kirchner (H.), « CARIBOO: An Induction Based Proof Tool for Termination with Strategies », dans *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming (PPDP)*, Pittsburgh, USA, octobre 2002. ACM Press.

[GG95] Grädel (E.) et Gurevich (Y.), « Tailoring Recursion for Complexity », *Journal of Symbolic Logic*, vol. 60, n° 3, septembre 1995, p. 952–969.

[Gie95] Giesl (J.), « Generating polynomial orderings for termination proofs », dans *Rewriting Techniques and Applications*, vol. 914 (coll. *Lecture Notes in Computer Science*), p. 427–431, 1995.

[Goe92] Goerdt (A.), « Characterizing complexity classes by higher type primitive recursive definitions », *Theoretical Computer Science*, vol. 100, n° 1, 1992, p. 45–66.

[Gur83] Gurevich (Y.), « Algebras of feasible functions », dans *Twenty Fourth Symposium on Foundations of Computer Science*, p. 210–214. IEEE Computer Society Press, 1983.

[Gur93] Gurevich (Y.), « Feasible Functions », *London Mathematical Society Newsletter*, n° 206, juin 1993, p. 6–7.

[Gur99] Gurevich (Y.), « The sequential ASM thesis », *EATCS Bulletin*, vol. 67, février 1999, p. 93–124.

[Göd31] Gödel (K.), « Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme », *Monatshefte für Mathematik und Physik*, vol. 38, 1931, p. 173–198.

[HL88] Hofbauer (D.) et Lautemann (C.), « Termination proofs and the length of derivations », dans *Rewriting Techniques and Applications*, vol. 355 (coll. *Lecture Notes in Computer Science*), 1988.

[Hof92] Hofbauer (D.), « Termination proofs with Multiset Path Orderings imply primitive recursive derivation lengths », *Theoretical Computer Science*, vol. 105, n° 1, 1992, p. 129–140.

[Hof97] Hofmann (M.), « A Mixed Modal/Linear Lambda Calculus with Applications to Bellantoni-Cook Safe Recursion », dans *CSL*, p. 275–294, 1997.

[Hof99] Hofmann (M.), « Linear types and Non-Size Increasing polynomial time computation », dans *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, p. 464–473, 1999.

[Hof02] Hofmann (M.), « The strength of Non-Size Increasing computation », dans *Proceedings of POPL'02*, p. 260–269, 2002.

[Hue80] Huet (G.), « Confluent reductions: Abstract properties and applications to term rewriting systems », *Journal of the ACM*, vol. 27, n° 4, 1980, p. 797–821.

[JGS93] Jones (N. D.), Gomard (C. K.) et Sestoft (P.), *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, juin 1993. Avec des chapitres par L. O. Andersen et T. Mogensen. Accessible depuis `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[Jon97] Jones (N. D.), *Computability and Complexity, from a Programming Perspective*. MIT press, 1997.

[Jon00] Jones (N. D.), « The Expressive Power of Higher-order Types or, Life without CONS », *Journal of Functional Programming*, vol. 11, n° 1, 2000, p. 55–94.

[Jon01] Jones (N. D.), « The Expressive Power of Higher order Types or, Life without CONS », *Journal of Functional Programming*, vol. 11, n° 1, janvier 2001, p. 55–94. Numéro spécial sur "Functionnal Programming and Computationnal Complexity".

[KB70] Knuth (D. E.) et Bendix (P. B.), « Simple word problems in universal algebras », *Computational Problems in Abstract Algebra*, 1970, p. 263–297.

[KK99] Kirchner (C.) et Kirchner (H.), « Rewriting, Solving, Proving ». Version préliminaire d'un livre accessible à `http://www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KL80] Kamin (S.) et Lévy (J.-J.), « Attempts for generalising the recursive path orderings. ». Rapport technique, Univerité de l'Illinois, Urbana, 1980. Note non publiée.

[Kle52] Kleene (S. C.), *Introduction to metamathematics*. North-Holland, 1952.

[Lan77] Lankford (D. S.), « Some approaches to equality for computational logic: A survey and assessment ». Rapport technique n° ATP-36, Automatic Theorem Proving Project, Université du Texas, 1977.

[Lan79] Lankford (D. S.), « On proving term rewriting systems are Noetherien ». Rapport technique n° MTP-3, Louisiana Technical University, 1979.

[Lar] *Larch*. `http://www.sds.lcs.mit.edu/spd/larch/index.html`.

[Lei94] Leivant (D.), « Predicative recurrence and computational complexity I: Word recurrence and poly-time », dans Clote (P.) et Remmel (J.), éditeurs, *Feasible Mathematics II*, p. 320–343. Birkhäuser, 1994.

[Lei99] Leivant (D.), « Ramified recurrence and computational complexity III: Higher type recurrence and elementary complexity », *Annals of Pure and Applied Logic*, vol. 96, n° 1-3, 1999, p. 209–229.

[Lie76] Lien (Y. E.), « Termination properties of generalized Petri Nets », *SIAM journal of computing*, vol. 5, n° 2, 1976, p. 251–265.

[LJBA01] Lee (C. S.), Jones (N. D.) et Ben-Amram (A. M.), « The Size-Change Principle for Program Termination », dans *Symposium on Principles of Programming Languages*, vol. 28, p. 81–92. ACM press, janvier 2001.

[LM93] Leivant (D.) et Marion (J.-Y.), « Lambda Calculus Characterizations of Poly-Time », *Fundamenta Informaticae*, vol. 19, n° 1,2, septembre 1993, p. 167–184.

[LM95] Leivant (D.) et Marion (J.-Y.), « Ramified recurrence and computational complexity II: Substitution and poly-space », dans Pacholski (L.) et Tiuryn (J.), éditeurs, *Computer Science Logic, 8th Workshop, CSL '94*, vol. 933 (coll. *Lecture Notes in Computer Science*), p. 486–500, Kazimierz, Pologne, 1995. Springer.

[LM97] Leivant (D.) et Marion (J.-Y.), « Predicative Functional Recurrence and Poly-Space », dans Bidoit (M.) et Dauchet (M.), éditeurs, *TAPSOFT'97, Theory and Practice of Software Development*, vol. 1214 (coll. *Lecture Notes in Computer Science*), p. 369–380. Springer, avril 1997.

[LM00a] Leivant (D.) et Marion (J.-Y.), « A characterization of alternating log time by ramified recurrence », *Theoretical Computer Science*, vol. 236, n° 1-2, avril 2000, p. 192–208.

[LM00b] Leivant (D.) et Marion (J.-Y.), « Predicative recurrence and computational complexity IV: Predicative functionals and poly-space », *Information and Computation*, 2000. À apparaître.

[LS99] Liu (Y. A.) et Stoller (S. D.), « Dynamic programming via static incrementalization. », dans *Proceedings of the 8th European Symposium on Programming*, p. 288–305, Amsterdam, Pays-Bas, mars 1999. Springer.

[Mar00a] Marion (J.-Y.), « Analysing the implicit complexity of programs », *Information and Computation*, 2000.

[Mar00b] Marion (J.-Y.), « Complexité implicite des calculs, de la théorie à la pratique », 2000. Habilitation.

[MM00] Marion (J.-Y.) et Moyen (J.-Y.), « Efficient First Order Functional Program Interpreter with Time Bound Certifications », dans Parigot (M.) et Voronkov (A.), éditeurs, *LPAR*, vol. 1955 (coll. *Lecture Notes in Artificial Intelligence*), p. 25–42. Springer, novembre 2000.

[MM03] Marion (J.-Y.) et Moyen (J.-Y.), « Termination and ressource analysis of assembly programs by Petri Nets ». Rapport technique, Loria, 2003.

[MN70] Manna (Z.) et Ness (S.), « On the termination of Markov algorithms », dans *Third hawaii international conference on system science*, p. 789–792, 1970.

[Moy01] Moyen (J.-Y.), « System Presentation: An analyser of rewriting systems complexity », dans van den Brand (M.) et Verma (R.), éditeurs, *Electronic Notes in Theoretical Computer Science*, vol. 59. Elsevier Science Publishers, 2001. Workshop RULE, accessible `http://www.loria.fr/~moyen/`.

[Pap94] Papadimitriou (C. H.), *Computational Complexity*. Addison-Wesley, 1994.

[Pét66] Péter (R.), *Rekursive Funktionen*. Akadémiai Kiadó, Budapest, Hongrie, 1966. Traduction anglaise : *Recursive Functions*, Academic Press, New York, 1967.

[Ric53] Rice (H. G.), « Classes of Recursively Enumerable Sets and Their Decision Problems. », *Trans. Amer. Math. Soc.*, vol. 74, 1953, p. 358–366.

[Ros84] Rose (H. E.), *Subrecursion*. Oxford university press, 1984.

[Sav70] Savitch (W. J.), « Relationship between nondeterministic and deterministic tape classes », *JCSS*, vol. 4, 1970, p. 177–192.

[Saz80] Sazonov (V.), « Polynomial computability and recursivity in finite domains », *Elektronische Informationsverarbeitung und Kybernetik*, vol. 7, 1980, p. 319–323.

[Sec02] Secher (J. P.), *Driving-based Program Transformation in Theory and Practice*. PhD thesis, DIKU, Université de Copenhague, 2002.

[Sim88] Simmons (H.), « The Realm of Primitive Recursion », *Archive for Mathematical Logic*, vol. 27, 1988, p. 177–188.

[Sim92] Simard (J.-F.), *Comment faire de la bonne bière chez soi*. Trécarré, 1992.

[Sko23] Skolem (T.), « Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Ausdehnungsbereich », *Skrifter utgit av Videnskapsselskapet i Kristiana,* I. *Matematisk-naturvidenskabelig klasse*, vol. 6, 1923, p. 3–38.

[Spr88] Spreen (D.), « On functions computable in nondeterministic polynomial time: Some characterizations », dans *Computer Science Logic*, vol. 329 (coll. *Lecture Notes in Computer Science*), p. 289–303. Springer, 1988.

[SS63] Shepherdson (J.) et Sturgis (H.), « Computability of recursive functions », *Journal of the ACM*, vol. 10, n° 2, 1963, p. 217–255.

[TG95] Turing (A. M.) et Girard (J.-Y.), *La machine de Turing*. Seuil, 1995.

[Thi93] Thibaut (P.), *Europa Universalis – Règles*. Azure Wish Edition, 1993.

[Tho72] Thompson (D.), « Subrecursiveness: machine independent notions of computability in restricted time and storage », *Math. System Theory*, vol. 6, 1972, p. 3–15.

[Tur36] Turing (A. M.), « On computable numbers with an application to the Entscheidungsproblem », *Proc. London Mathematical Society*, vol. 42, n^{o} 2, 1936, p. 230–265. Traduction [TG95].

[Tur86] Turchin (V. F.), « The concept of a supercompiler », *Transactions on Programming Languages and Systems*, vol. 3, n^{o} 8, 1986, p. 292–325.

[Wad90] Wadler (P.), « Deforestation: transforming programs to eliminate trees », *Theoretical Computer Science*, vol. 73, n^{o} 2, 1990, p. 231–248.

[Wei95] Weiermann (A.), « Termination proofs by Lexicographic Path Orderings yield multiply recursive derivation lengths », *Theoretical Computer Science*, vol. 139, 1995, p. 335–362.

[WMO89] Watanabe (T.), Mizobata (Y.) et Onaga (K.), « Minimum Initial Marking Problems of Petri Nets », *Transactions of the Institute of Electronics, Information and Communication Engineers*, vol. E72, n^{o} 12, décembre 1989, p. 1390–1399.

Résumé

Le thème de la thèse concerne l'analyse automatique de la complexité des programmes, en particulier en terme de complexité de la fonction calculée et non de l'algorithme implémenté.

D'une part, la notion d'ordres de terminaison des systèmes de réécriture est restreinte à l'aide de quasi-interprétations, ce qui permet de donner une borne sur la complexité de la fonction calculée. Cette borne n'est pas nécessairement atteinte par le système étudié et il peut être nécessaire de transformer le programme pour l'atteindre. Une caractérisation de PTIME et une de PSPACE sont ainsi obtenues.

D'autre part, un mini langage d'assemblage est étudié au moyen de réseaux de Petri. Il devient ainsi possible de regrouper dans une seule analyse une preuve de terminaison proche du "Size-Change Principle", une preuve de calcul en place (sans allouer de mémoire supplémentaire) et une simplification du programme similaire à la déforestation. De plus, cette technique permet de prouver la terminaison d'une très grande classe de programmes. En particulier, la terminaison d'algorithmes comme celui d'Euclide ou comme les algorithmes "Diviser pour régner" est obtenue de manière totalement automatique.

Mots-clés : Terminaison, complexité, réécriture, réseaux de Petri, déforestation.

Abstract

This thesis is about automatic analysis of the complexity of programs, especially the complexity of the function computed rather than the complexity of the algorithm implemented.

First, termination orderings are restricted by the mean of quasi-interpretations, allowing one to give a bound on the complexity of the computed function. The system itself may compute in a time significantly larger than this bound, so it may be necessary to automatically transform the system in order to achieve the bound. In this way, a characterisation of both PTIME and PSPACE are obtained.

Then, a small assembly-like language is studied via Petri nets. This allows a single analysis to do at the same time a termination proof close to the Size-Change Principle, a proof of the Non-Size Increasingness of the program and a simplification similar to Deforestation. Moreover, this technique is able to prove the termination of a wide class of programs. Especially, the termination of algorithms such as Euclid's one or the "Divide and Conquer" algorithms is obtained in a fully automated way.

Keywords : Termination, Complexity, Rewriting, Petri nets, Deforestation.