NANCY UNIVERSITE

IUT A, département informatique — **École doctorale IAE + M**

Département de formation doctorale en informatique

Complexité implicite des calculs, de la théorie à la pratique

MÉMOIRE

Vendredi 15 Décembre 2000

pour l'obtention de l'

Habilitation de l'Université Nancy 2

(Spécialité Informatique)

par

Jean-Yves Marion

Composition du jury

Rapporteurs :	T. Coquand
	E. Grandjean
	N. Jones
	C. Kirchner
Examinateurs :	A. Cichon
	S. Grigorieff (directeur)
	D. Leivant
	M. Margenstern
	J. Souquières

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

Loria

Mis en page avec la classe thloria.

Le sujet de ce mémoire est la complexité implicite des calculs. Il a fallu faire des choix, et nous avons essayé de faire un exposé cohérent qui montre l'évolution du thème de la théorie vers les applications. En contrepartie, ce mémoire n'est pas un état de l'art complet du domaine. Les démonstrations sont, en grande partie, omises.

D'autre part, le mémoire ne couvre qu'une partie de mes recherches. Ainsi, mon travail de thèse sur la complexité des fonctions définies sur des modèles finis [Mar91], mes travaux sur la logique linéaire [BM96, GM95a, GM95b, Mar96, Mar97, Mar99] et sur la complexité en information [GM00] n'apparaissent pas.

Table des matières

Introduction

1 Une problématique

La construction de logiciels sûrs est une nécessité. Il est tout aussi crucial dans le développement d'un logiciel certifié de s'assurer de la qualité de l'implantation en terme d'efficacité et de ressources de calcul.

La méta-théorie de la programmation répond traditionnellement à des questions de correction par rapport à une spécification, comme la terminaison. Ces propriétés sont dites extensionnelles. Cependant, certaines propriétés, comme l'efficacité d'un programme et les ressources employées pour effectuer un calcul, sont exclues de cette méthodologie. La cause de cette lacune tient à la nature des questions posées. Dans le premier cas, nous traitons une propriété extensionnelle, tandis que dans le second cas, nous abordons la question sur la manière dont la fonction est réalisée et comment un calcul est effectué. Dès lors, nous nous intéressons à une propriété intensionnelle[1] des programmes. Pourtant, la maîtrise de ces facteurs permet de s'assurer de la qualité d'une implantation.

Le sujet de ce mémoire est de présenter des méthodes qui, à partir d'un programme, analyse la complexité de la fonction calculée par ce programme. A l'aide des informations recueillies par cette analyse, il est possible de construire un programme plus efficace. Qu'est-ce que cela veut dire ?

La complexité d'un programme est une mesure des ressources nécessaires à son exécution. Les ressources qui sont prises en compte sont, usuellement, le temps et l'espace. La théorie de la complexité étudie les problèmes et les fonctions qui sont calculables avec une certaine quantité de ressources. Il ne faut pas confondre la complexité d'une fonction avec la complexité d'un programme. Une fonction est réalisée par différents programmes. Certains programmes sont efficaces, d'autres ne le sont pas.

Un succès de la théorie de la complexité est de préciser à "l'expert en programmation" les limites de son art, et ceci quels que soient les giga-octets et les méga-flops à sa disposition. Un autre succès de la théorie de la complexité est de fournir un modèle mathématique de la complexité algorithmique. Mais face à ces modèles, l'expert en programmation est en plein désarroi. Les causes en sont diverses, illustrons-les par deux exemples. Le théorème d'accélération linéaire affirme que tout programme qui s'exécute en temps $T(n)$ (où n est la taille de l'entrée) peut être transformé en un programme équivalent qui calcule en temps $\epsilon T(n)$ où ϵ est aussi petit que nous voulons. Ce résultat n'a

[1] Le mot intension est admis dans le Petit Robert, à ne pas confondre avec intention.

aucune contrepartie *réelle*. Un autre exemple qui est plus en relation avec ce mémoire est le suivant. Une fonction est faisable si elle est calculée par un programme dont la complexité est acceptable. La classe des fonctions faisables est souvent identifiée avec la classe PTIME des fonctions calculables en temps polynomial. Un résultat typique est de définir un langage de programmation L et de démontrer que la classe de fonctions calculées par les programmes de L, *i.e.* $\{[\![\mathtt{p}]\!]_{\mathtt{L}} : \mathtt{p}$ est un programme de $\mathtt{L}\}$, est exactement la classe PTIME. Ce type de résultat ne répond pas aux questions de l'expert en programmation, car le langage de programmation L ne contient pas les "bons algorithmes", qu'il utilise quotidiennement. Le fossé entre les deux disciplines s'explique encore par une différence de point de vue. La théorie de la complexité, fille de la théorie de la calculabilité, a gardé un point de vue extensionnel, dans la modélisation, tandis que la théorie de la programmation[2] est génétiquement intensionnelle.

Pourquoi ce long discours ? La nécessité de raisonner sur les programmes est une question pertinente dans le processus de développement des logiciels. La certification d'un programme est une propriété essentielle, mais elle n'est pas la seule. Démontrer la terminaison d'un programme de complexité exponentielle n'a pas de sens par rapport à notre réalité. Il se pose alors le problème de la construction d'outils pour raisonner sur les algorithmes. Face à ce vaste chantier, notre contribution se porte à un coin du grand puzzle de la théorie des algorithmes, que nous nommons *complexité implicite des calculs*. Il s'agit d'analyser la complexité des algorithmes.

2 Plan chapitre par chapitre

Nous résumons chaque chapitre de ce mémoire. Chaque résumé est repris, pour l'essentiel, en début de chapitre. Après la lecture du premier chapitre, le lecteur pourra aborder tous les chapitres, à deux exceptions près. Le chapitre 5 dépend du chapitre 4, et le chapitre final dépend du chapitre 4 et de la fin du chapitre 8.

Chap. 1 Un Langage fonctionnel du premier ordre : E

Ce chapitre est consacré à la description du langage de programmation E qui nous suivra au long de ce mémoire. E est un langage fonctionnel typé du premier ordre. Le domaine de calcul comprend les structures habituelles comme les mots, les listes, les arbres. Les programmes sont écrits sous forme d'équations. La sémantique calculatoire est basée sur la réécriture de termes. E s'apparente à Elan et au fragment du premier ordre de Haskell ou de ML, par exemple.

Chap. 2 Langages de programmation et modèles de calcul

Nous proposons une mesure du temps de calcul d'un programme de E. Pour cela, nous munissons E d'une horloge *Clock* telle que $\mathit{Clock}_{\mathtt{f}}(\vec{\mathbf{a}})$ est égale au nombre minimal d'étapes de réductions pour atteindre la forme normale de

[2]Non, il n'y a pas de contradiction avec le début de l'introduction. La théorie de la programmation traite des algorithmes, de leurs constructions et de leurs transformations, alors que la méta-théorie de la programmation étudie et justifie la théorie de la programmation, afin, par exemple, de certifier la correction des programmes.

$\mathtt{f}(\vec{\mathbf{a}})$. A partir de cette horloge, nous définissons DTIME($T(n)$) comme l'ensemble des fonctions calculables en moins de $T(n)$ étapes par un programme de $\mathtt{E}$ où n est la somme des tailles des entrées. La classe des fonctions calculables en temps polynomial s'exprime, alors, par

$$\text{PTIME} = \cup_{T \text{ est un polynôme}} \text{DTIME}(T(n))$$

En préambule, nous discuterons des modèles de calcul et nous tenterons de justifier le choix de la mesure du temps associée à $\mathtt{E}$, en nous reposant sur le livre de Jones [Jon97]. Nous serons conduits à parler de la complétude extensionnelle et des complétudes intensionnelles.

Chap. 3 Interprétations des définitions par récurrence

Nous abordons les rapports entre la conception axiomatique des entiers et les algorithmes. Par la suite, seules les définitions d'itérations primitives seront utiles. Nous pensons présenter un point de vue sémantique sur la complexité algorithmique, qui est original, bien que l'argumentation soit d'une grande simplicité. Les sources d'inspiration sont le livre de Kleene [Kle52] et l'article de Henkin [Hen60].

Chap. 4 Analyse prédicative de la récurrence

Nous abordons le chapitre central de ce mémoire. L'analyse prédicative de la récurrence provient des travaux de Bellantoni et Cook [BC92] et de Leivant [Lei94c]. Cette analyse repose sur un principe de ramification dont l'attrait provient de sa simplicité conceptuelle. Chaque terme du calcul a une valence qui détermine sa capacité à exécuter une récurrence. Le principe de ramification affirme qu'une définition par récurrence est ramifié seulement si la valence associée au paramètre de récurrence est strictement supérieure à la valence de l'argument critique. De façon imagée, une valence est un niveau d'énergie et un calcul par récurrence consomme de l'énergie. Cette perte d'énergie est traduite par les valences associées. Cette analyse a pour grands frères l'article de Simmons [Sim88] et celui de Leivant [Lei91].

Nous nous concentrerons sur les fonctions calculables en temps polynomial, et nous listons d'autres caractérisations en fin de chapitre. Nous détaillerons et comparons les approches de Bellantoni et Cook, et de Leivant. Nous avons apporté une nouvelle contribution qui est une caractérisation des classes DTIME($O(n^k)$).

Chap. 5 Au-dessus de l'exponentielle

Ce chapitre est une digression pour expliquer comment les différentes approches de l'analyse prédicative s'insèrent dans la hiérarchie de Grzegorczyk. Bien que les fonctions auront une complexité démesurée, il nous semble que ces études permettent de mieux comprendre l'analyse prédicative et son rapport aux fondements des mathématiques.

Dans un premier temps, nous partirons de la construction d'Ackermann pour définir une fonction qui n'est pas primitive récursive. Nous verrons comment diagonaliser les définitions par récurrence ramifiée soit de façon externe en suivant les travaux de Caporaso, Pani et Covino soit de façon interne comme nous le proposons.

Ensuite, nous expliquerons comment Leivant a ramifié le système T de Gödel et la relation avec les itérateurs de Church du λ-calcul simplement typé.

Nous terminerons avec la définition de mesure fine du degré de récurrence proposée par Bellantoni et Niggl, dans la tradition de Heinermann [Hei61], Schwichtenberg [Sch69] et Müller [Mül74]. Notre guide pour ce chapitre a été le livre de Simmons [Sim00].

Chap. 6 Synthèse de programmes efficaces en théories des types

Une théorie logique H définit un ensemble, récursivement énumérable, de fonctions totales. Ainsi, l'arithmétique de Peano définit les fonctions programmées par le système T (Gödel [Göd58]), et l'arithmétique du second ordre définit les fonctions programmées par le système F (Girard [Gir72]). Existe-t-il une théorie logique des fonctions calculables en temps polynomial ? Buss [Bus86], Leivant [Lei91] et d'autres ont construit de tels systèmes logiques. Dans ce chapitre, nous proposons une restriction de l'arithmétique de Peano, qui caractérise les fonctions calculables en temps polynomial. Cette restriction porte sur la quantification universelle qui est limitée à une catégorie de termes, appelée canonique. Le formalisme s'inspire de l'arithmétique fonctionnelle du second ordre AF_2 de Leivant [Lei83]. Ce résultat est une contribution originale de ce mémoire. En contrepartie, ce chapitre est plus ardu.

Une motivation est la question, un peu flou, d'appréhender ce que pourrait être les mathématiques attachées à la réalité du temps polynomial, comme il y a les mathématiques constructives. Plus concrètement, la théorie des types de Martin-Löf [ML84], le calcul des constructions [CH88] sont des théories logiques à partir duquel il est possible de synthétiser des programmes sûrs à partir d'une démonstration de leurs corrections. Dans cette perspective, notre apport est la possibilité de certifier la complexité du programme synthétisé.

Chap. 7 Ordre et terminaison

Un programme termine si son calcul s'arrête sur toutes entrées. Pour démontrer la terminaison d'un programme, une méthode consiste à définir un ordre bien fondé sur les termes, et à prouver qu'à chaque étape de réduction le terme obtenu décroît. Bien que ce petit chapitre traite d'aspects fondamentaux, il n'est pas essentiel à la bonne compréhension de la suite. Nous verrons comment Dershowitz a énoncé, à partir du théorème de Higman-Kruskal, des conditions suffisantes pour qu'un ordre sur les termes démontre la terminaison d'un programme. La plupart des ordres utilisés dans la pratique vérifient ces propriétés. Ces ordres sont appelés ordres de simplification et nous donnerons des exemples dans le chapitre suivant. Notre guide pour ce chapitre a été l'état de l'art de Gallier [Gal91] et celui de Dershowitz [Der87].

Chap. 8 Expressivité et ordres de simplification

Au chapitre précédent, nous avons défini une classe d'ordres, les ordres de simplification, dont les propriétés étaient suffisantes pour montrer la terminaison d'un programme.

Ce chapitre présente et examine l'expressivité d'ordres de simplification bien connus et utilisés. Nous entendons par expressivité d'un ordre, la complexité des programmes de `E` qui terminent par cet ordre. Nous verrons que dans le cas des preuves de terminaison par interprétation polynomiale, les programmes sont effectivement calculables. Plus exactement, nous caractérisons

les classes PTIME, DTIME($2^{O(n)}$) et DTIME($2^{2^{O(n)}}$). En conséquence, les preuves par interprétation polynomiale peuvent déterminer la complexité pratique d'un programme. D'un autre côté, l'expressivité des ordres récursifs sur les chemins est énorme. Nous capturons les fonctions primitives récursives et les fonctions multiples récursives, suivant la manière de comparer les suites d'arguments. L'intérêt de ces ordres est pleinement justifié par le fait qu'ils capturent des schémas d'algorithmes utiles dans la pratique.

Chap. 9 Vers une analyse des algorithmes

Nous sommes arrivés au centre de la problématique de l'introduction qui est l'analyse des algorithmes. Nous montrons comment les principes qui régissent l'analyse prédicative et les ordres de terminaison, permettent de déterminer la complexité d'une fonction à partir d'un programme, *i.e.* la complexité implicite de la fonction. Une conséquence est la possibilité de transformer le programme en un programme plus efficace. Nous verrons que dans certains cas, le programme avec sa sémantique usuelle se calcule en temps exponentiel, alors que la fonction qu'il dénote est calculable en temps polynomial, par un autre algorithme.

3 Résultats nouveaux

Ce mémoire contient quelques résultats orignaux et non publiés.

1. Le chapitre 3 suggère une nouvelle approche sémantique.
2. Une caractérisation de DTIME(n^k) dans le chapitre 4 avec des types strictement ramifiés.
3. La méthode de diagonalisation avec des types dépendants du chapitre 5.
4. Le chapitre 6 ne contient que des résultats originaux.
5. Dans le chapitre 9, nous avons intégré le statut lexicographique, ce qui est une amélioration de l'article de Marion [Mar00] et de Marion et Moyen [MM00b].
6. Toujours dans le chapitre 9, nous avons minimisé les valeurs stockées dans le cache de l'interpréteur.

Première partie

Programmes et calculs

Chapitre 1

Un langage fonctionnel du premier ordre : E

Nous commençons par la description du langage de programmation E qui nous suivra au long de ce mémoire. E est un langage fonctionnel typé du premier ordre. Le domaine de calcul comprend les mots, les listes, les arbres. Les programmes sont écrits sous forme d'équations. La sémantique calculatoire est basée sur la réécriture de termes. E s'apparente à Elan et au fragment du premier ordre de Haskell ou de ML, par exemple.

1.1 Syntaxe des programmes

Le domaine de calcul et la syntaxe des programmes de E sont construits comme suit :

Définition 1.1. Un programme de E est un quadruplet $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ dont la syntaxe est décrite dans la figure 1.1, et où

- $\mathcal{C}$ est le vocabulaire fini des constructeurs avec arités.
- $\mathcal{F}$ est le vocabulaire fini des symboles de fonctions avec arités, disjoint de $\mathcal{C}$.
- $\mathcal{X}$ est le vocabulaire des variables, disjoint de $\mathcal{C} \cup \mathcal{F}$.
- $\mathcal{E}$ est un ensemble d'équations orientées. Chaque variable qui apparaît dans le membre droit d'une équation, apparaît aussi à gauche. La partie gauche l d'une équation $l \to r \in \mathcal{E}$ est un schéma de redex. Les schémas redex considérés dans les programmes ont une forme particulière puisque $l = \mathtt{f}(p_1, \ldots, p_n)$ où f est le symbole défini et le motif p_i appartient à $\mathcal{T}(\mathcal{C}, \mathcal{X})$. Enfin, on supposera que les variables de p_i sont distinctes des variables de p_j, pour tout $i \neq j$.

Nous écrirons $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ pour indiquer que f est le symbole de fonction principal du programme. Par abus de langage, le symbole de fonction principal f désignera aussi le programme qui le définit.

L'ensemble des variables d'un terme t est $var(t)$. Pour des raisons typographiques, nous abrègerons $x_1, \ldots, x_n$ par $\vec{x}$.

Exemple 1.2. Le programme suivant renverse une liste. L'ensemble des constructeurs est $\mathcal{C} = \{\mathbf{nil}, \mathbf{cons}\}$, l'ensemble des symboles de fonctions est $\mathcal{F} =$

(Constructeurs)	$\mathcal{C} \ni \mathbf{c}$	::=	$\mathbf{c} \mid \mathbf{b} \mid \ldots$
(Symboles de fonction)	$\mathcal{F} \ni \mathtt{f}$	::=	$\mathtt{f} \mid \mathtt{g} \mid \mathtt{h} \mid \ldots$
(Variables)	$\mathcal{X} \ni x$	::=	$x \mid y \mid z \mid \ldots$
(Termes constructeurs)	$\mathcal{T}(\mathcal{C}) \ni \mathbf{a}$	::=	$\mathbf{c}(\mathbf{a}_1)\ldots(\mathbf{a}_n) \mid \mathbf{b}$
(Termes clos)	$\mathcal{T}(\mathcal{C},\mathcal{F}) \ni s$	::=	$\mathbf{c}(s_1)\ldots(s_n) \mid \mathtt{f}(s_1)\ldots(s_n)$
(Termes)	$\mathcal{T}(\mathcal{C},\mathcal{F},\mathcal{X}) \ni t$	::=	$x \mid \mathbf{c}(t_1)\ldots(t_n) \mid \mathtt{f}(t_1)\ldots(t_n)$
(Motifs)	$\mathcal{T}(\mathcal{C},\mathcal{X}) \ni p$	::=	$\mathbf{c}(p_1)\ldots(p_n) \mid x$
(Equations)	$\mathcal{E} \ni e$	::=	$\mathtt{f}(p_1)\ldots(p_n) \to t$

FIG. 1.1 – Grammaire des termes

$\{\mathtt{renv}, \mathtt{renverser}\}$, et `renverser` est le symbole de fonction principal. Enfin, l'ensemble $\mathcal{E}$ comporte les trois équations ci-dessous.

$$\begin{aligned} \mathtt{renv}(\mathbf{nil}, r) &\to r \\ \mathtt{renv}(\mathbf{cons}(a, l), r) &\to \mathtt{renv}(l, \mathbf{cons}(a, r)) \\ \mathtt{renverser}(x) &\to \mathtt{renv}(x, \mathbf{nil}) \end{aligned}$$

1.2 Types

Définition 1.3. Soit $\mathcal{S}$ un ensemble de sortes. L'ensemble des types est

(*Atomes*)	σ	::=	$\mathbf{s} \in \mathcal{S}$
(*Types*)	τ	::=	$\sigma \to \tau \mid \sigma$

Un type non-atomique, $\mathbf{s}_1 \to (\ldots(\mathbf{s}_n \to \mathbf{s}))$ est le type d'une fonction. Pour insister sur la restriction au premier ordre, nous écrirons $\mathbf{s}_1, \ldots, \mathbf{s}_n \to \mathbf{s}$.

Définition 1.4. Un environnement de typage Γ est une fonction, à domaine fini, de $\mathcal{C} \cup \mathcal{F}$ dans *Types*. Un environnement de typage des variables Θ est une fonction, à domaine fini, de $\mathcal{X}$ dans *Atomes*.

Les équations de typage sont données dans la figure 1.2. Un terme t est de type τ dans les environnements Γ et Θ si $\Gamma, \Theta \vdash t : \tau$. Par la suite, quand les environnements de typage sont suffisamment clairs, ils ne seront pas mentionnés et nous noterons juste $t : \tau$.

En particulier, si $\Gamma(\mathbf{d}) = \mathbf{s}$ alors $\mathbf{d}$ est un constructeur d'arité 0. Si $\Gamma(\mathtt{g}) = \mathbf{s}_1, \ldots, \mathbf{s}_n \to \mathbf{s}$, alors $\mathtt{g}$ est un symbole d'arité n, et nous préférons l'écriture $\mathtt{g}(t_1, \ldots, t_n)$ à la notation à la Curry $\mathtt{g}(t_1)\ldots(t_n)$.

Définition 1.5. Un programme typé $\mathtt{f}$ est un quintuplet $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \Gamma \rangle$ constitué d'un programme $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ et d'un environnement de typage Γ. Pour chaque équation $\mathtt{f}(p_1, \ldots, p_n) \to t$, nous déduisons un environnement de variables Θ tel que le membre gauche et le membre droit de l'équation ont le même type, *i.e.* $\Gamma, \Theta \vdash \mathtt{f}(p_1, \ldots, p_n) : \tau$ et $\Gamma, \Theta \vdash t : \tau$.

Remarque 1.6. L'environnement des variables Θ est local car le type des variables est déterminé par le type des constructeurs et des symboles de fonctions. Toutes les règles de typage correspondent à des règles d'élimination en

$$\textit{(Symbole de fonction)} \qquad \frac{\Gamma(\mathtt{f}) = \tau}{\Gamma, \Theta \vdash \mathtt{f} : \tau}\,(\mathtt{f} \in \mathcal{F})$$

$$\textit{(Constructeur)} \qquad \frac{\Gamma(\mathbf{c}) = \tau}{\Gamma, \Theta \vdash \mathbf{c} : \tau}\,(\mathbf{c} \in \mathcal{C})$$

$$\textit{(Variable)} \qquad \frac{\Theta(x) = \mathbf{s}}{\Gamma, \Theta \vdash x : \mathbf{s}}\,(x \in \mathcal{X})$$

$$\textit{(Application)} \qquad \frac{\Gamma, \Theta \vdash s : \mathbf{s} \quad \Gamma, \Theta \vdash t : \mathbf{s} \to \tau}{\Gamma, \Theta \vdash t(s) : \tau} \to E$$

FIG. 1.2 – Équations de typage des termes

déduction naturelle. Les environnements de typage jouent le rôle des règles d'introduction.

Exemple 1.7. Reprenons l'exemple 1.2. Nous avons

$$\Gamma(\mathtt{renv}) = \mathbf{list}(\mathbf{s}), \mathbf{list}(\mathbf{s}) \to \mathbf{list}(\mathbf{s})$$
$$\Gamma(\mathtt{renverser}) = \mathbf{list}(\mathbf{s}) \to \mathbf{list}(\mathbf{s})$$

où $\mathbf{s}$ est une sorte[3].

1.3 Domaines de calcul

La calculabilité étudie les fonctions sur le domaine des entiers, et la théorie de la complexité sur les mots. Notre étude des algorithmes et de la complexité nous amène à considérer des domaines de calcul plus expressifs. Les programmes de `E` ont pour domaine de calcul des algèbres de termes qui représentent les entiers bâtons, les mots, les listes de mots, les arbres, ... Chaque domaine de calcul est défini par une structure composée d'une sorte et des constructeurs qui l'engendrent[4]. Nous exposons les structures des principaux domaines de calcul que nous rencontrerons dans la suite.

Les booléens

La structure des booléens est $\langle \mathbf{bool}\ ,\ \mathbf{tt} : \mathbf{bool},\ \mathbf{ff} : \mathbf{bool} \rangle$.

Les entiers bâtons (unaires)

La structure des entiers bâtons est $\langle \mathbf{nat},\ \mathbf{0} : \mathbf{nat},\ \mathbf{suc} : \mathbf{nat} \to \mathbf{nat} \rangle$. La sorte **nat** permet de représenter un entier n en notation unaire $\underline{n}$ par $\underline{0} = \mathbf{0}$ et $\underline{n+1} = \mathbf{suc}(\underline{n})$.

[3] Cet exemple illustre la capacité du langage `E` à maîtriser un polymorphisme paramétrique.

[4] Et, bien entendu, nous confondrons la structure et la sorte.

Les mots

La structure des mots binaires est $\langle \mathbf{word},\ \boldsymbol{\epsilon}:\mathbf{word},\ \mathbf{0}:\mathbf{word},\ \mathbf{1}:\mathbf{word}\rangle$. La structure **word** code les mots de $\{0,1\}^*$ par $\underline{\epsilon} = \boldsymbol{\epsilon}$ et $\underline{iu} = \mathbf{i}(\underline{u})$.

Le produit cartésien

La structure du produit cartésien de type $\mathbf{s}$ est $\langle \mathbf{record_k}(\mathbf{s})\ ,\ [\,]:\mathbf{s}^k \to \mathbf{s}\rangle$.

Les listes

La structure des listes de type $\mathbf{s}$ est

$$\langle \mathbf{list}(\mathbf{s}),\ \mathbf{nil}:\mathbf{list}(\mathbf{s}),\ \mathbf{cons}:\mathbf{s},\mathbf{list}(\mathbf{s}) \to \mathbf{list}(\mathbf{s})\rangle$$

Remarque 1.8. Dans la présentation faite, les sortes ne sont pas des types, mais le domaine des structures algébriques sur lequel s'effectue un calcul. Nous aurions pu avoir un point de vue "théorie des types". Pour cela, définissons $\mathbf{1}$ comme le type unité, i.e. qui contient un seul élément. Prenons deux constructeurs de types, à savoir $\times$ le produit cartésien et $+$ l'union disjointe. Un type récursif T est défini par la plus petite solution d'une équation de la forme $T = F[T]$, où F est la définition du type construite avec $+$ (union disjointe) et $\times$ (produit cartésien), par exemple. Ainsi, les booléens sont définis par la règle $\mathbf{bool} = \mathbf{1} + \mathbf{1}$. Les entiers par $\mathbf{nat} = \mathbf{1} + \mathbf{nat}$, et enfin les listes d'objets de type A sont définies par $\mathbf{list}(A) = \mathbf{1} + A \times \mathbf{list}(A)$.

1.4 Calculs et sémantique

La sémantique de E s'appuie sur les réductions engendrées par les équations du programme. Nous rappelons quelques définitions familières à la théorie de la réécriture. Pour un état de l'art, nous renvoyons à l'article de Dershowitz et Jouannaud [DJ90] ou au livre de C. Kirchner et H. Kirchner [KK].

Nous écrirons $t[u_1,\ldots,u_n]$ pour indiquer que les u_i sont des sous-termes qui ont des occurrences disjointes. Le terme $t[u_1 \leftarrow s_1,\ldots,u_n \leftarrow s_n]$ est obtenu en remplaçant simultanément chaque u_i par s_i.

Une *substitution* (resp. *substitution close*) σ est une fonction, à domaine fini, des variables dans les termes de $\mathcal{T}(\mathcal{C},\mathcal{F},\mathcal{X})$ (resp. $\mathcal{T}(\mathcal{C},\mathcal{F})$). Le terme $t\sigma$ est le terme $t[x_1 \leftarrow s_1,\ldots,x_n \leftarrow s_n]$ où $\sigma = \{x_1 \leftarrow s_1,\ldots,x_n \leftarrow s_n\}$. Le terme u est un redex s'il existe une équation $l \to r$ de $\mathcal{E}$ et une substitution σ telle que $u = l\sigma$. La relation $\to$ est étendue aux termes en construisant la plus petite relation qui est monotone et stable par substitution.

(Monotonie)

$$\text{Si } u \to s \text{ alors } t[u] \to t[u \leftarrow s]$$

(Stable par substitution)

$$\text{Pour chaque substitution } \sigma, \text{ si } t \to s \text{ alors } t\sigma \to s\sigma$$

Notons $\xrightarrow{+}$ (resp. $\xrightarrow{*}$) la clôture transitive (resp. réflexive et transitive) de la relation de réduction $\to$. Enfin, t a pour forme normale s si $t \xrightarrow{*} s$ et s ne contient pas de redex. Nous écrirons alors $t \xrightarrow{!} s$.

L'interprétation d'un programme est formée de l'interprétation des sortes et de l'interprétation des termes. L'interprétation $[\![\mathbf{s}]\!]$ de la sorte $\mathbf{s}$ est l'ensemble des termes constructeurs de type $\mathbf{s}$, i.e. $[\![\mathbf{s}]\!] = \{\mathbf{a}:\mathbf{s},\ \text{et } \mathbf{a} \in \mathcal{T}(\mathcal{C})\}$. Les

formes normales significatives appartiennent aux interprétations des sortes. L'interprétation d'un terme est un peu plus délicate. En effet, un terme peut ne pas avoir de forme normale quand aucune réduction partant de t ne termine. Ou encore, un terme peut posséder une ou plusieurs formes normales. Ces deux observations mettent en valeur deux propriétés fondamentales des systèmes de réécriture qui sont la *terminaison* et la *confluence*. L'existence d'une forme normale pour chaque terme est assurée quand le système de réécriture termine. L'unicité de la forme normale d'un terme, quand elle existe, est assurée quand le système est confluent, ce qui est équivalent à dire que le système possède la propriété de Church-Rosser.

Définition 1.9. La relation $\rightarrow$ est confluente si pour tous termes z, x et y, tels que $z \xrightarrow{*} x$ et $z \xrightarrow{*} y$, alors il existe un terme t tel que $x \xrightarrow{*} t$ et $y \xrightarrow{*} t$.

Le programme est confluent si la relation de réduction associée $\rightarrow$ est confluente.

Remarque 1.10. La confluence est une propriété indécidable, en général. Cependant, il existe des conditions suffisantes pour s'assurer de la confluence d'un programme. Ainsi, un programme qui vérifie les deux conditions ci-dessous est appelé *orthogonal*. Un programme orthogonal est confluent, voir l'article de Huet [Hue80].

Linéarité gauche Pour chaque équation $l \rightarrow r$, le terme l est linéaire, c'est à dire chaque variable de l n'a qu'une seule occurrence dans l.

Non ambiguïté Soient $l_1 \rightarrow r_1$ et $l_2 \rightarrow r_2$ deux équations. Il n'existe pas de substitutions σ_1 et σ_2 telles que $\sigma_1(l_1) = \sigma_2(l_2)$.

La condition d'orthogonalité est souvent vérifiée en programmation.

Un programme confluent calcule une fonction partielle.

Définition 1.11. Supposons que $\mathtt{f}$ soit de type $\mathbf{s}_1, \ldots, \mathbf{s}_n \rightarrow \mathbf{s}$. Le programme confluent $\mathtt{f}$ est interprété par la fonction $[\![\mathtt{f}]\!] : [\![\mathbf{s}_1]\!] \times \ldots [\![\mathbf{s}_n]\!] \rightarrow [\![\mathbf{s}]\!]$ qui est définie ainsi : $[\![\mathtt{f}]\!](\mathbf{a}_1, \ldots, \mathbf{a}_n) = \mathbf{d}$ si, et seulement si, $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n) \xrightarrow{!} \mathbf{d}$, pour tout $\mathbf{a}_i \in [\![\mathbf{s}_i]\!]$ et $\mathbf{d} \in [\![\mathbf{s}]\!]$.

Remarque 1.12. Par extension, $[\![t]\!]$ est égal à la forme normale du terme t, quand celle-ci existe. Nous aurions pu interpréter les termes par un élément de $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})/\stackrel{*}{\leftrightarrow}$. Dans ce cas, chaque terme divergent a une dénotation différente.

Dans la suite, *nous restreindrons aux programmes qui (i) sont confluents et (ii) terminent*, sauf mention explicite du contraire comme au chapitre 8.

Chapitre 2

Langages de programmation et modèles de calcul

Nous proposons une mesure du temps de calcul d'un programme de $\mathtt{E}$. Pour cela, nous munissons $\mathtt{E}$ d'une horloge *Clock* telle que $\mathit{Clock}_{\mathtt{f}}(\vec{\mathtt{a}})$ est égale au nombre minimal d'étapes de réductions pour atteindre la forme normale de $\mathtt{f}(\vec{\mathtt{a}})$. A partir de l'horloge, nous définissons $\mathrm{DTIME}(T(n))$ comme l'ensemble des fonctions calculables en moins de $T(n)$ étapes par un programme de $\mathtt{E}$ où n est la somme des tailles des entrées et $T : \mathbb{N} \to \mathbb{N}$ est une fonction totale. La classe des fonctions calculables en temps polynomial s'exprime, alors, par

$$\mathrm{PTIME} = \bigcup_{T \text{ est un polynôme}} \mathrm{DTIME}(T(n))$$

En préambule, nous discuterons des modèles de calcul et nous tenterons de justifier le choix de la mesure du temps associée à $\mathtt{E}$. Nous serons conduits à parler de la complétude extensionnelle et des complétudes intensionnelles.

2.1 Langages de programmation

En suivant de nombreux auteurs, un langage de programmation $\mathtt{L}$ se décompose en :

1. un ensemble de données $\mathcal{D}(\mathtt{L})$ ou domaine de calcul,
2. un ensemble de programmes $\mathcal{P}(\mathtt{L})$,
3. une fonction sémantique $[\![\ ]\!]_{\mathtt{L}}$ qui associe à chaque programme une fonction partielle. La classe des fonctions calculées est

$$\mathcal{F}(\mathtt{L}) = \{[\![\mathtt{p}]\!]_{\mathtt{L}} \ : \ \mathtt{p} \in \mathcal{P}(\mathtt{L})\}$$

Cette découpe s'applique au langage fonctionnel $\mathtt{E}$ du chapitre 1, mais aussi au λ-calcul où l'ensemble des programmes et des données est identique. C'est aussi un simple exercice de définir un langage de programmation qui calcule toutes les fonctions récursives, et dont le domaine de calcul est celui des entiers écrits en unaire.

Un langage de programmation exprime la notion d'*algorithme* à un certain niveau d'abstraction. Or, ce niveau d'abstraction peut-être élevé. Est-ce qu'un

langage de programmation est une modélisation satisfaisante de la notion de calcul ? Une propriété désirable d'un modèle de calcul est d'être suffisamment proche de la réalité physique et qu'une étape de calcul soit, en quelque sorte, atomique. Une étape de calcul d'un programme de `E` est une réduction qui consiste à remplacer une occurence de redex dans un terme. Le temps de calcul est alors la longueur de la dérivation. Est-ce bien raisonnable ? Il serait, peut être préférable d'associer, à chaque étape de réduction, un coût proportionnel à la hauteur du terme. Nous voyons qu'un moyen de contourner ces difficultés serait de définir un modèle de calcul de référence, et d'interpréter les langages dans ce modèle.

2.2 Machines et ressources

Turing [Tur36, TG95] a proposé un modèle de calcul en terme de *machines*. Il a démontré que les différentes définitions des fonctions calculables sont bien calculables par machines de Turing. Ce faisant, Turing a clairement confirmé la thèse de Church[5], car il a relié le monde mathématique et la calculabilité à une vision discrète de la réalité physique et à une notion concrète du calcul.

Depuis, divers modèles de calcul ont été proposés, qui sont décrits dans tous les livres sur la calculabilité et la complexité. Parmi les modèles de machines déterministes et séquentielles, il y a les Random Access Machines [SS63] (`RAM`), les machines de Kolmogorov-Uspensky [KU58, Gur88], les Abstract State Machines (`ASM`) de Gurevich [Gur99]. Tous ces modèles ont au moins trois points communs :

1. une notion de configuration avec des configurations initiales et finales,
2. un nombre fini de règles qui permettent de passer d'une configuration à une autre,
3. le passage d'une configuration à une autre qui nécessite une quantité fixe d'information.

Mais, il ne faut pas s'y tromper et comme dans le cas des langages de programmation, le niveau d'abstraction des algorithmes dépend du modèle de machine. L'inégalité est flagrante entre une machine de Turing et une machine `RAM`. Cette dernière accède directement aux registres, additionne deux nombres en 1 étape, ...

A la différence des langages de programmation, le concept de ressource est attaché au fonctionnement des machines. La mesure des ressources valide l'adéquation entre le modèle de calcul et une réalité quotidienne.

Fixons un modèle de machine `M` qui sera notre mètre étalon. Un bon choix serait les machines `RAM` qui ne disposent que de la seule opération arithmétique $n \mapsto n + 1$. Il faut prendre garde à la puissance de calcul du modèle de machine choisi. Par exemple, les automates cellulaires dans le plan hyperbolique de Margenstern et Morita [MM00a], résolvent le problème SAT en temps polynomial.

Disons qu'un *langage de programmation* `L` *est raisonnable* s'il existe un interpréteur de `L` dans `M` qui exécute les programmes de `L`. Cette définition

[5] L'article de Sieg [Sie97] retrace la genèse de la thèse de Church.

nous permet de mesurer les ressources d'un calcul de L. Ainsi, un langage de programmation raisonnable sera muni d'une horloge. Pour simplifier, nous supposerons que les programmes sont d'arité 1. Une horloge est une fonction $Clock^{\mathtt{L}} : \mathcal{P}(\mathtt{L}) \to \mathcal{D}(\mathtt{L}) \to \mathbb{N} \cup \{\bot\}$ telle que $Clock^{\mathtt{L}}_{\mathtt{p}}(\mathbf{a})$ est le temps du calcul de p sur l'entrée $\mathbf{a}$ avec le modèle de calcul fixé.

2.3 Les complétudes

Soit X une classe de fonctions. Lorsque toutes les fonctions de X sont calculées par un programme de L, et réciproquement, *i.e.* $X = \mathcal{F}(\mathcal{P}(\mathtt{L}))$, alors L est complet par rapport à X. Nous dirons que cette *complétude est extensionnelle* car elle est relative aux fonctions calculées. Typiquement, la thèse de Church énonce un résultat de complétude extensionnelle. Mais, il y a d'autres formes de complétudes liées aux algorithmes.

Commençons par la *complétude intensionnelle relative*. Pour simplifier, prenons deux langages de programmation raisonnables S et L qui ont le même ensemble de données, *i.e.* $\mathcal{D}(\mathtt{S}) \subseteq \mathcal{D}(\mathtt{L})$ et $\mathcal{P}(\mathtt{S}) \subseteq \mathcal{D}(\mathtt{L})$. Un interpréteur $\mathtt{int}_{\mathtt{S}\Rightarrow\mathtt{L}}$ de S dans L est un programme de L tel que pour tout programme p de S, $\mathtt{int}_{\mathtt{S}\Rightarrow\mathtt{L}}(\mathtt{p})$ est un programme de L tel que $[\![\mathtt{int}_{\mathtt{S}\Rightarrow\mathtt{L}}(\mathtt{p})]\!]_{\mathtt{L}} = [\![\mathtt{p}]\!]_{\mathtt{S}}$.

Remarque 2.1. Cette définition d'un interpréteur inclut la possibilité de spécialiser un argument. Autrement dit, les fonctions s^n_m de Kleene doivent être définissables dans L.

La condition $\mathcal{P}(\mathtt{S}) \subseteq \mathcal{D}(\mathtt{L})$ est une commodité pour indiquer que la syntaxe concrète des programmes de $\mathcal{P}(\mathtt{S})$ est dans $\mathcal{D}(\mathtt{L})$.

Maintenant, nous dirons que L est complet par rapport à S s'il existe une constante[6] c et un interpréteur $\mathtt{int}_{\mathtt{S}\Rightarrow\mathtt{L}}$ tel que pour tout programme p de S,

$$\forall \mathbf{a} \in \mathcal{D}(\mathtt{L}), \quad Clock^{\mathtt{L}}_{\mathtt{int(p)}}(\mathbf{a}) \leq c \cdot Clock^{\mathtt{S}}_{\mathtt{p}}(\mathbf{a})$$

La complétude intensionnelle relative est utilisée pour comparer les langages de programmation raisonnables. Par exemple, les machines de Turing avec 1 seule bande ne sont pas complètes, dans le sens sus-cité, par rapport aux machines de Turing avec 2 bandes. Pourquoi ? Le problème de la reconnaissance d'un palindrome se résout en $2n + O(1)$ étapes avec 2 bandes. D'un autre côté, ce problème nécessite au moins n^2 étapes avec 1 bande. Cette borne inférieure peut être obtenue par un argument d'incompressibilité [PSS81] qui utilise la complexité de Kolmogorov [LV97].

Une notion plus forte est celle de *complétude intensionnelle absolue*. Un langage de programmation L possède cette propriété si L est intensionnellement complet relativement à tout autre langage de programmation. En quelque sorte, L est *universel pour les algorithmes*. Cette complétude est développée par Gurevich [Gur99], et dans lequel il énonce sa thèse :

Les ASM sont universels pour les algorithmes.

Tout comme la thèse de Church, l'existence d'un langage de programmation qui est universel pour les algorithmes, n'est pas démontrable, car la notion

[6] La constante ne dépend pas de p.

d'algorithme n'est pas formelle. Cependant, nous pouvons chercher des conditions nécessaires pour cette universalité.

Disons qu'un self-interpréteur `int` de `L` est un interpréteur de `L` dans `L`. Un self-interpréteur est efficace si il existe une constante c telle que pour tout programme `p` de `L`,

$$\forall \mathbf{a} \in \mathcal{D}(\mathtt{L}), \quad \mathit{Clock}^{\mathtt{L}}_{\mathtt{int(p)}}(\mathbf{a}) \leq c \cdot \mathit{Clock}^{\mathtt{L}}_{\mathtt{p}}(\mathbf{a})$$

Si `L` est universel pour les algorithmes, alors il possède nécessairement un interpréteur efficace. En effet, `L` est universel relativement à lui même. L'existence d'un self-interpréteur efficace est une propriété fondamentale qui est au cœur de deux beaux résultats. Le premier, dû à Levin [Lev73, Gur88], est la construction d'un algorithme optimal pour inverser les fonctions. Le second, dû à Jones [Jon93], est la démonstration de l'existence d'une hiérarchie stricte des langages reconnus en temps linéaire, en dépit du théorème d'accélération linéaire des machines de Turing que nous avons cité en introduction.

2.4 Temps de calcul associé à `E`

$$\frac{\sigma(x) = \mathbf{e}}{\mathcal{E}, \sigma \vdash x \underset{1}{\rightarrow} \mathbf{e}}$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash t_i \underset{n_i}{\longrightarrow} \mathbf{d}_i}{\mathcal{E}, \sigma \vdash \mathbf{c}(t_1, \ldots, t_n) \xrightarrow[\sum_{i=1,n} n_i]{} \mathbf{c}(\mathbf{d}_1, \ldots, \mathbf{d}_n)}$$

$$\frac{\mathcal{E}, \sigma \vdash t_i \underset{n_i}{\longrightarrow} \mathbf{d}_i \quad \mathtt{f}(\vec{p}) \rightarrow r \in \mathcal{E} \quad p_i\sigma' = \mathbf{d}_i \quad \mathcal{E}, \sigma' \vdash r \underset{n}{\rightarrow} \mathbf{e}}{\mathcal{E}, \sigma \vdash \mathtt{f}(t_1, \ldots, t_n) \xrightarrow[n+\sum_{i=1,n} n_i]{} \mathbf{e}}$$

FIG. 2.1 – Interpréteur par appel par valeur, étant donné un programme $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \Gamma \rangle$, et une assignation σ des variables

.

Nous avons décidé de prendre un coût unitaire pour une réduction. Le temps d'exécution d'un programme `f` sur les entrées $\vec{\mathbf{a}}$ est le nombre de pas de réductions nécessaires pour atteindre la forme normale de $\mathtt{f}(\vec{\mathbf{a}})$, suivant l'interpréteur par appel par valeur qui décrit dans la figure 2.1. Nous avons $\mathcal{E}, \sigma \vdash t \underset{r}{\rightarrow} v$ si le terme $t\sigma$ se réduit en r étapes dans sa forme normale v, en suivant les équations $\mathcal{E}$.

A partir de là, l'horloge *Clock* associée à `E` est définie par

$$\mathit{Clock}_{\mathtt{f}}(\mathbf{a}_1, \ldots, \mathbf{a}_m) = \min_{\theta}\{r : \mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_m) \underset{r}{\rightarrow} \mathbf{d} \text{ et } [\![\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_m)]\!] = \mathbf{d}\}$$

Cette horloge est justifiée par le langage `WHILE` qui est utilisé dans le livre de Jones [Jon97]. En effet, `WHILE` est intensionnellement complet par rapport

à E. Mais, comme nous l'avons déjà dit, ce choix est discutable. Regardons l'exemple suivant qui recopie trois fois un arbre binaire.

$$\mathtt{B_3}(x) \rightarrow \mathbf{cons}(x, \mathbf{cons}(x, x))$$

Si **a** est un arbre binaire, $\mathtt{B_3}(\mathbf{a})$ se réduit en une seule étape. Ce coût unitaire se justifie si les structures de données arborescentes sont représentées par des graphes acycliques. Dans notre exemple, le calcul revient à créer deux cellules **cons** et les faire pointer vers **a**. En pratique, c'est ce qui est fait. Nous ne détaillerons pas plus, et nous renvoyons au livre de Jones [Jon97] (voir chapitre 16 à 18, et en particulier, au langage similaire F).

2.5 Les classes DTIME($T(n)$) et PTIME

La taille $|t|$ d'un terme t est le nombre de symbole qui compose t. Pour terminer, définissons l'ensemble des programmes dont le temps de calcul est borné par $T(n)$

$$\mathtt{E}^{time(T(n))} = \{\mathtt{f} \,:\, \forall \vec{\mathbf{a}} \in \mathcal{T}(\mathcal{C}), \mathit{Clock}_{\mathtt{f}}(\vec{\mathbf{a}}) \leq T(n), \text{ où } n = \sum_{1 \leq i \leq m} |\mathbf{a}_i|\}$$

Ensuite, la classe des fonctions calculables en temps $T(n)$ est définie par

$$\text{DTIME}(T(n)) = \{[\![\mathtt{f}]\!] \,:\, \mathtt{f} \in \mathtt{E}^{time(T(n))} \text{ où } \mathtt{f} \in \mathcal{P}(\mathtt{E})\}$$

La classe des fonctions calculables en temps polynomial est

$$\text{PTIME} = \cup_{T \text{ est un polynôme}} \text{DTIME}(T(n))$$

Posons $\exp_0(n) = n$ et $\exp_{k+1}(n) = 2^{\exp_k(n)}$. Nous verrons aussi des fonctions dont le temps de calcul est borné par $\exp_k(n)$.

Pour le reste, nous nous reportons aux livres de Papadimitriou [Pap94], Jones [Jon97] et Savage [Sav98] sur la complexité algorithmique.

Chapitre 3

Interprétations des définitions par récurrence

Nous abordons les rapports entre la conception axiomatique des entiers et les algorithmes. Par la suite, seules les définitions d'itérations primitives seront utiles. Nous pensons présenter un point de vue sémantique sur la complexité algorithmique, qui est original, bien que l'argumentation soit d'une grande simplicité. Les sources d'inspiration sont le livre de Kleene [Kle52] et l'article de Henkin [Hen60]. Enfin, bien que nous nous soyons essentiellement concentrés sur les entiers, les idées et les résultats s'étendent, facilement, à toutes les structures de données.

3.1 Conception axiomatique des entiers

Dans cette section, nous ne considérons que des programmes de `E` sur la sorte **nat**. Cependant, il n'y a pas de difficulté à généraliser la discussion à n'importe qu'elle sorte.

3.1.1 Interprétations

Une *interprétation* $(N, 0, S)$ de la structure $\langle \mathbf{nat},\ \mathbf{0}:\mathbf{nat},\ \mathbf{suc}:\mathbf{nat}\to\mathbf{nat}\rangle$ consiste en un ensemble N, un élément 0, et une opération successeur, unaire, S sur N. La sémantique canonique, sur l'algèbre libre des termes constructeurs, décrite en 1.4, forme l'interprétation $([\![\mathbf{nat}]\!], \mathbf{0}, \mathbf{suc})$. Cette interprétation est isomorphe à l'interprétation $(\mathbb{N}, 0, \lambda n.n+1)$, où $\mathbb{N}$ est l'ensemble des entiers et $\lambda n.n+1$ est la fonction successeur. Par la suite, nous désignerons par *interprétation standard* toute interprétation isomorphe à $([\![\mathbf{nat}]\!], \mathbf{0}, \mathbf{suc})$, et nous raisonnerons à un isomorphisme près. Cela aura pour conséquence que nous passerons de $([\![\mathbf{nat}]\!], \mathbf{0}, \mathbf{suc})$ à l'interprétation $(\mathbb{N}, 0, \lambda n.n+1)$, sans prévenir le lecteur.

Un homomorphisme entre deux interprétations $(N', 0', S')$ et $\mathcal{I} = (N, 0, S)$ est une fonction $[\![\ ]\!]_{\mathcal{I}} : N' \to N$ telle que $[\![0']\!]_{\mathcal{I}} = 0$ et $[\![S'(x)]\!]_{\mathcal{I}} = S([\![x]\!]_{\mathcal{I}})$. L'interprétation des termes de $\langle \mathbf{nat},\ \mathbf{0}:\mathbf{nat},\ \mathbf{suc}:\mathbf{nat}\to\mathbf{nat}\rangle$ dans $\mathcal{I}$ peut-être vue comme l'homomorphisme $[\![\ ]\!]_{\mathcal{I}}$ entre l'interprétation standard et $\mathcal{I}$.

Théorème 3.1. *Soit $\mathcal{I}$ une interprétation. Il existe un unique homomorphisme $[\![\ ]\!]_{\mathcal{I}}$ entre l'interprétation standard et $\mathcal{I}$. Autrement dit, l'interprétation standard est initiale.*

3.1.2 Modèles

Toutes les interprétations ne sont pas isomorphes et toutes les interprétations ne sont pas des modèles du langage de programmation `E`. En effet, posons $N_n = \{0, \ldots, n\}$, et définissons la fonction successeur comme $T_n(m) = (m+1) \bmod (n+1)$. Considérons le programme `add` qui additionne deux nombres, dans le modèle standard.

$$\begin{aligned} \texttt{add}(\mathbf{0}, y) &\to y \\ \texttt{add}(\mathbf{suc}(x), y) &\to \mathbf{suc}(\texttt{add}(x, y)) \end{aligned}$$

L'addition modulo $(n+1)$ est l'unique fonction sur N_n qui vérifie les équations de `add`. Maintenant, prenons le programme exp qui définit la fonction exponentielle en base 2.

$$\begin{aligned} \texttt{exp}(\mathbf{0}) &\to \mathbf{suc}(\mathbf{0}) \\ \texttt{exp}(\mathbf{suc}(x)) &\to \texttt{add}(\texttt{exp}(x), \texttt{exp}(x)) \end{aligned}$$

Il n'y a pas de fonction qui vérifie les équations de l'exponentielle sur le domaine N_2, car nous aurions à la fois $exp(0) = 1$, et $exp(0) = 0$. En effet, $exp(0) = [\![[\![\texttt{exp}]\!](\mathbf{0})]\!]_{N_2} = 1$, par la première équation de `exp`, et $exp(0) = [\![[\![\texttt{exp}]\!](\mathbf{suc}^2(\mathbf{0})]\!]_{N_2} = 0$ en utilisant la deuxième équation et en remarquant que $\mathbf{suc}^2(\mathbf{0})$ est interprété par 0.

Soit `f` un programme d'arité 1. La section 1.4 décrit une sémantique calculatoire qui associe à `f` une fonction $[\![\texttt{f}]\!] : [\![\mathbf{nat}]\!] \to [\![\mathbf{nat}]\!]$. Une interprétation $M = (N, 0, S)$ est un modèle d'un programme, compatible avec la sémantique calculatoire, s'il existe une fonction f sur N telle que pour tout $\mathbf{a} \in [\![\mathbf{nat}]\!]$, nous avons

$$\begin{array}{ccc} [\![\mathbf{nat}]\!] & \xrightarrow{[\![f]\!]} & [\![\mathbf{nat}]\!] \\ \downarrow{\scriptstyle [\![\ ]\!]_M} & & \downarrow{\scriptstyle [\![\ ]\!]_M} \\ N & \xrightarrow{f} & N \end{array} \qquad f([\![\mathbf{a}]\!]_M) = [\![[\![\texttt{f}]\!](\mathbf{a})]\!]_M$$

Ainsi, $(N_n, 0, T_n)$ est un modèle de `add`. Un modèle de `E` est un modèle pour tous les programmes. L'interprétation standard $(\mathbb{N}, 0, \lambda n.n+1)$ est un modèle de `E`, tandis que $(N_n, 0, T_n)$ ne l'est pas. En fait, les modèles de `E` sont tous isomorphes à l'interprétation standard, comme nous allons le voir.

3.1.3 Modèle de Peano

En suivant Henkin [Hen60], nous dirons qu'une interprétation $(N, 0, S)$ est un *modèle de Peano* s'il vérifie ces trois axiomes :

(P1) Pour tout $x \in N$, $S(x) \neq 0$.

(P2) Pour tout $x, y \in N$, si $x \neq y$ alors $S(x) \neq S(y)$.

(P3) Si G est un sous-ensemble de N tel que (a) $0 \in G$ et (b) $x \in G$ implique $S(x) \in G$, alors (c) $G = N$.

L'interprétation standard $(\mathbb{N}, 0, \lambda n.n+1)$ est un modèle de Peano. L'interprétation $(N_n, 0, T_n)$ n'est pas un modèle de Peano, car il satisfait seulement les axiomes (P2) et (P3), mais pas (P1), Plus généralement, les interprétations dont le domaine est fini, ne sont pas des modèles de Peano.

Théorème 3.2. *Tous les modèles de Peano sont isomorphes.*

Remarque 3.3. On dit aussi que les axiomes (P1),(P2),(P3) sont catégoriques. Les axiomes de Peano constituent une définition des entiers au second ordre. Il n'y a pas d'axiomatique catégorique au premier ordre.

3.2 Algorithmes primitifs récursifs

3.2.1 Fonctions récursives primitives

Les fonctions primitives récursives ont été introduites par Skolem en 1927. Les deux références incontournables sont les livres de Péter [Pét66] et de Rose [Ros84]. L'importance des fonctions primitives récursives provient de leur relation intime avec le schème des entiers, tel qu'il est défini par les axiomes (P1),(P2) et (P3). La classe des fonctions primitives récursives est la plus petite classe de fonctions qui contient la fonction nulle, la fonction successeur, les projections, et qui est close par composition et par récursion primitive.

3.2.2 Programmes primitifs récursifs

Dans son livre [Kle52], Kleene présente les fonctions primitives récursives par un langage de programmation, PR(**nat**), défini comme suit. Le domaine de calcul est $\langle \mathbf{nat},\ \mathbf{0}:\mathbf{nat},\ \mathbf{suc}:\mathbf{nat}\to\mathbf{nat}\rangle$. Les programmes sont construits par *définition explicite* et par **nat**-*récursion*.

Définition explicite Un symbole de fonction `f` est défini explicitement

$$\mathtt{f}(p_1, \ldots, p_n) \to t$$

si chaque symbole de fonction dans t est déjà défini par une équation du programme.

Exemple 3.4. Le prédecesseur est obtenu ainsi,

$$\begin{aligned} \mathtt{pred}(\mathbf{0}) &\to \mathbf{0} \\ \mathtt{pred}(\mathbf{suc}(x)) &\to x \end{aligned}$$

nat-récursion Un symbole de fonction `f` est défini par **nat**-récursion

$$\begin{aligned} \mathtt{f}(\mathbf{0}, \vec{y}) &\to \mathtt{g}(\vec{y}) \\ \mathtt{f}(\mathbf{suc}(x), \vec{y}) &\to \mathtt{h}(x, \vec{y}, \mathtt{f}(x, \vec{y})) \end{aligned}$$

si `g` et `h` sont déjà définis dans le programme.

Le premier argument de $\mathtt{f}$ sera appelé le *paramètre de la récurrence*, et $\mathtt{f}(x, \vec{y})$ sera appelé l'*argument critique* dans $\mathtt{h}$.

Théorème 3.5. *La classe de fonctions* $\mathcal{F}(\mathrm{PR}(\mathbf{nat}))$ *est exactement la classe des fonctions primitives récursives.*

Le théorème suivant est une justification méta-mathématique de la récursion primitive par le principe d'induction sur les entiers.

Théorème 3.6. *Les modèles des programmes de* $\mathrm{PR}(\mathbf{nat})$ *sont des modèles de Peano.*

Par conséquent, tous les modèles de $\mathtt{E}$ sont des modèles de Peano, et donc sont isomorphes.

3.2.3 Itérations sur les entiers

Le langage de programmation $\mathrm{SPR}(\mathbf{nat})$ est défini comme suit. Le domaine de calcul est $\langle \mathbf{nat},\ \mathbf{0}:\mathbf{nat},\ \mathbf{suc}:\mathbf{nat}\to\mathbf{nat}\rangle$. Les programmes sont construits par *définition explicite* et par **nat**-*itération*.

nat-itération Un symbole de fonction $\mathtt{f}$ est défini par **nat**-itération

$$\mathtt{f}(\mathbf{0}, \vec{y}) \to \mathtt{g}(\vec{y})$$
$$\mathtt{f}(\mathbf{suc}(x), \vec{y}) \to \mathtt{h}(\vec{y}, \mathtt{f}(x, \vec{y}))$$

si $\mathtt{g}$ et $\mathtt{h}$ sont déjà définis dans le programme.

Gladstone [Gla67] a démontré que la récursion primitive se réduit à l'itération.

Théorème 3.7. *La classe de fonctions* $\mathcal{F}(\mathrm{SPR}(\mathbf{nat}))$ *est exactement la classe des fonctions primitives récursives.*

3.2.4 Itérations sur une sorte

Le schéma d'itération se généralise à toutes les sortes. Le langage de programmation $\mathrm{SPR}(\mathbf{s})$ a pour domaine de calcul la structure engendrée par $\mathbf{s}$ et les programmes sont construits par *définition explicite* et par $\mathbf{s}$-*itération*.

s-itération Un symbole de fonction $\mathtt{f}$ est défini par $\mathbf{s}$-itération

Pour chaque constante $\mathbf{b}:\mathbf{s}$, il y a une équation

$$\mathtt{f}(\mathbf{b}, \vec{y}) \to \mathtt{g}_{\mathbf{b}}(\vec{y})$$

de même, pour chaque constructeur $\mathbf{c}:\mathbf{s}_1, \ldots, \mathbf{s}_n \to \mathbf{s}$,

$$\mathtt{f}(\mathbf{c}(x_1, \ldots, x_n), \vec{y}) \to \mathtt{h}_{\mathbf{c}}(\vec{y}, \mathtt{f}(x_{i_1}, \vec{y}), \ldots \mathtt{f}(x_{i_m}, \vec{y})) \qquad \text{où } x_{i_j}:\mathbf{s}$$

si $\mathtt{g}$ et $\mathtt{h}$ sont déjà définis dans le programme [7].

Théorème 3.8. *Supposons que* $[\![\mathbf{s}]\!]$ *soit infini. La classe de fonctions* $\mathcal{F}(\mathrm{SPR}(\mathbf{s}))$ *est exactement la classe des fonctions primitives récursives.*

[7] Il y a une petite difficulté qui est due aux structures hétérogènes comme les listes. Il faut comprendre que $\mathbf{s}_{i_j} = \mathbf{s}$.

3.2.5 Itérations avec substitutions

Un symbole de fonction $\mathtt{f}$ est défini par itération avec substitutions de paramètres si

$$\mathtt{f}(\mathbf{0}, \vec{y}) \to \mathtt{g}(\vec{y}) \tag{3.1}$$
$$\mathtt{f}(\mathbf{suc}(x), \vec{y}) \to \mathtt{h}(\vec{y}, t_1, \ldots, t_m) \tag{3.2}$$

avec $t_j = \mathtt{f}(x, \sigma_0^j(\vec{y}), \ldots, \sigma_n^j(\vec{y}))$, pour tout $1 \leq j \leq m$.

La classe de programmes SRSP(**nat**) est l'ensemble des programmes pour lesquels chaque symbole de fonction est soit défini explicitement soit défini par itération avec substitutions de paramètres.

Théorème 3.9. *La classe de fonction* $\mathcal{F}(\mathrm{SRSP}(\mathbf{nat}))$ *est exactement la classe des fonctions primitives récursives.*

3.2.6 Le comportement des algorithmes

Les théorèmes ci-dessus énoncent un résultat de nature extensionelle, *e.g.* $\mathcal{F}(\mathrm{SRSP}(\mathbf{nat})) = \mathcal{F}(\mathrm{SPR}(\mathbf{nat}))$. Cependant, la traduction d'une itération avec substitution de paramètres en une simple itération a un coût. Est-ce que les programmes de SPR(**nat**) sont aussi efficaces que ceux de SRSP(**nat**) ? Les travaux de Colson sur le comportement des algorithmes abordent cette difficile question. Pour un tour d'horizon de ses travaux, on consultera [Col98, Col99]. Colson a démontré que la fonction *inf*, qui calcule le minimum de deux entiers, n'est pas programmable en temps $O(\min(n, m))$ dans PR(**nat**). Or, Valarcher [Val96] a remarqué qu'il y a un programme de SRSP(**nat**) qui calcule *inf* en temps $O(\min(n, m))$. Plus simplement, le programme $\mathtt{inf}$ décrit ci-dessous, qui n'est pas de SRSP(**nat**), calcule efficacement *inf* :

$$\mathtt{inf}(\mathbf{0}, y) \to \mathbf{0}$$
$$\mathtt{inf}(x, \mathbf{0}) \to \mathbf{0}$$
$$\mathtt{inf}(\mathbf{suc}(x), \mathbf{suc}(y)) \to \mathbf{suc}(\mathtt{inf}(x, y))$$

Ces remarques soulèvent la question de la complétude intensionnelle par rapport à la classe des fonctions primitives récursives. Quels sont les algorithmes qui définissent des fonctions primitives récursives ? Le livre de Péter [Pét66] est un catalogue de tels algorithmes. Dans l'article [Sim88], Simmons a essayé de comprendre la nature de ces algorithmes.

Plus généralement, étant donnés deux langages Turing-complets, quel est le meilleur langage ? JAVA ou ML ? Comment les comparer ? L'étude comparative des schémas de programme essaie de répondre à ces questions. La schématologie remonte aux années 70 avec les travaux de Paterson. On pourra consulter l'état de l'art écrit par Courcelle [Cou90]. La comparaison s'effectue en interprétant les schémas de programme au moyen de sémantiques dénotationnelles, de jeux (*Pebble game*), de modèles finis.

3.3 Axiomes de Peano et complexité

3.3.1 Modèles finis

Nous allons faire un détour par la théorie descriptive de la complexité. Le récent livre d'Immerman [Imm99] offre un large tour d'horizon de cette discipline. En deux mots, cette théorie étudie les propriétés des interprétations de cardinalité finie qui sont exprimées par des formules logiques. Typiquement, pour parler de propriétés sur les graphes, nous ajoutons un prédicat unaire S pour désigner les sommets et un prédicat binaire A pour les arcs. Une interprétation de cardinalité finie est la donnée d'un graphe particulier. Les modèles d'une formule logique sont exactement les interprétations de cardinalité finie qui vérifient la propriété exprimée par la formule. Un exemple typique est la propriété d'être un "graphe hamiltonien", ou encore d'être un "graphe complet". Un résultat représentatif est celui de Fagin [Fag74], dans lequel il a démontré que les propriétés exprimables dans le fragment existentiel de la logique du second ordre caractérisent NPTIME, *i.e.* exactement toutes les propriétés calculables en temps polynomial sur machine de Turing non-déterministe.

Un ingrédient fondamental de la théorie descriptive de la complexité est le caractère fini des interprétations considérées. Cette restriction s'applique aux langages de programmation. Ainsi, un programme est interprété uniformément sur la famille d'interprétations finies $(N_n, 0, S_n)_{n\in\mathbb{N}}$, où l'opération successeur est $S_n(m) = m + 1$ si $m < n$ et $S_n(n + i) = n$. Un programme `f` définit une fonction $[\![\mathtt{f}]\!]_n$ sur chaque interprétation $(N_n, 0, S_n)$ telle que $[\![\mathtt{f}]\!]_n(m) = [\![\mathtt{f}]\!](m)$, pour tout $m \in N_n$. Ensuite, nous définissons une fonction globale F telle que $F(n) = [\![\mathtt{f}]\!]_n$. Nous étudions, alors, la complexité des fonctions globales calculées par un langage de programmation. Parmi les résultats obtenus, nous avons :

- La classe des fonctions globales qui sont calculées par un programme de PR($\mathbf{record_k}$($\mathbf{nat}$)) est exactement la classe LOGSPACE des fonctions calculables en espace logarithmique, résultat dû à Gurevich [Gur83].
- La classe des fonctions globales calculées par un programme quelconque de `E`($\mathbf{record_k}$($\mathbf{nat}$)) est exactement la classe PTIME des fonctions calculables en temps polynomial, résultat dû à Sazonov [Saz80] et Gurevich [Gur83].
- La classe des fonctions globales calculées par un programme du système T de Gödel caractérise les classes DTIME($exp_k(n)$) et DSPACE($exp_k(n)$) où k dépend du degré de fonctionnalité du programme, résultat dû à Goerdt [Goe92a, Goe92b].

Remarque 3.10. La complexité est mesurée par rapport au cardinal de la structure.

Plus récemment, dans l'article [Jon99], qui est repris dans le chapitre 24 de son livre [Jon97], Jones a étudié un langage de programmation, dit *cons-free*. Un tel langage ne peut pas effectuer un `cons` dans une assignation. C'est à dire, une assignation de la forme `X :=cons(A,L)` est interdite. Les seules assignations autorisées utilisent des destructeurs, *e.g.* `X :=tail(L)`. Jones a établi :

- Les langages reconnus par un programme cons-free de PR(**list**(**nat**))

sont exactement les langages décidables en espace logarithmique, résultat dû à Jones [Jon99, Jon00].
- Les langages reconnus par un langage de programmation WHILE cons-free sont exactement les langages décidables en temps polynomial, résultat dû à Jones [Jon99].
- Les langages reconnus par programme du système T de Gödel caractérisent les classes de langages DTIME($exp_k(n)$) et DSPACE($exp_k(n)$) où k dépend du degré de fonctionnalité du programme, résultats dûs à Jones [Jon00].

La similarité entre les approches "globale" et "cons-free" a été observée par Jones. Dans l'approche "globale", la modélisation de la complexité algorithmique s'appuie sur des modèles de cardinalité finie, qui satisfont les axiomes (P1) et (P3), mais pas l'axiome (P2)[8]. L'approche "cons-free" s'appuie sur une restriction syntaxique des programmes. Est-ce qu'une interprétation qui vérifie seulement (P1) et (P3) est un modèle d'un programme cons-free ?

Remarque 3.11. Traditionnellement, la théorie de la complexité descriptive étudie des modèles qui satisfont (P1) et (P3). Comme l'a remarqué Gurevich et comme cela est fait dans la thèse de Marion [Mar91], nous pourrions considérer des modèles qui satisfont plutôt (P2) et (P3).

3.3.2 Modèles faibles

Un modèle faible est un modèle d'un programme qui n'est pas un modèle de Peano. Le modèle $(N_n, 0, T_n)$ est un modèle faible puisqu'il ne vérifie pas l'axiome (P1). Pourtant, $(N_n, 0, T_n)$ est un modèle de l'addition `add`. La théorie de la complexité descriptive repose sur des modèles faibles, qui ne vérifient pas (P2).

En fait, l'existence et l'unicité d'une fonction qui vérifie les équations de `add` sont impliquées par l'existence et l'unicité d'un homomorphisme entre deux interprétations $(N, 0, S)$ et $(N', 0', S')$, défini par les équations ci-dessous.

$$\mathtt{h}_y(\mathbf{0}) \to \mathbf{0'}$$
$$\mathtt{h}_y(\mathbf{suc}(x)) \to \mathbf{suc'}(\mathtt{h}_y(x))$$

L'addition est définissable par recollement sur y, en posant $\mathtt{add}(x, y) = h_y(x)$ et en interprétant $\mathbf{0'}$ par y. Le théorème 3.1 affirme qu'il existe un unique homomorphisme entre un modèle de Peano et n'importe quelle autre interprétation $(N', 0', S')$. Par conséquent, l'addition est définissable de N vers N'. Il en serait, de même, pour la multiplication.

Nous voyons ainsi qu'il n'est pas nécessaire d'employer des modèles de Peano pour démontrer l'existence et l'unicité d'une fonction, comme dans le cas de l'addition.

Quelle est la relation entre les classes de programme de faible complexité (sous-exponentielle) et les modèles faibles ? Les axiomes vérifiés par une interprétation inséminent des propriétés algorithmiques aux entiers. Un modèle de Peano conçoit les entiers dans leur totalité. La démonstration de l'existence de

[8] Il y a des interprétations infinies qui satisfont (P1) et (P3), mais pas (P2).

la fonction exponentielle nécessite que l'argument critique $\texttt{exp}(n)$ soit interprété dans un modèle de Peano. Bien entendu, cela n'a rien de choquant d'un point de vue mathématique. Maintenant, pour l'informaticien, il faut justifier que les ressources disponibles sont suffisantes au calcul $\texttt{exp}(n)$.

Peut-être à tort ou par ignorance de l'auteur, mais il semble que les modèles faibles donnent un nouvel éclairage aux différentes modélisations de la complexité algorithmique, et aux idées de Nelson [Nel86], reprises par Leivant dans plusieurs de ses articles. Pour terminer, revenons sur les modèles constitués de deux interprétations $(N, 0, S)$ et $(N', 0', S')$. Chacune de ces interprétations vérifie des axiomes, ce qui implique une certaine conception algorithmique des entiers. Cette distinction entre le domaine et l'image peut être traduite dans la syntaxe des programmes en utilisant des copies de la structure $\langle \mathbf{nat},\ \mathbf{0}:\mathbf{nat},\ \mathbf{suc}:\mathbf{nat}\to\mathbf{nat}\rangle$. Nous obtenons une stratification des entiers suivant leurs propriétés algorithmes, et cela sera un fil conducteur des chapitres suivants.

Deuxième partie

Introduction à la complexité implicite des calculs

Chapitre 4

Analyse prédicative

Nous abordons le chapitre central de ce mémoire. L'analyse prédicative de la récurrence provient des travaux de Bellantoni et Cook [BC92] et de Leivant [Lei94c]. Cette analyse repose sur un principe de ramification dont l'attrait provient de sa simplicité conceptuelle. Chaque terme du calcul a une valence qui détermine sa capacité à exécuter une récurrence. Le principe de ramification affirme qu'une définition par récurrence est ramifiée seulement si la valence associée au paramètre de récurrence est strictement supérieure à la valence de l'argument critique. De façon trés imagée, une valence est un niveau d'énergie et un calcul par récurrence consomme de l'énergie. Quand un argument n'a plus d'énergie, il ne peut plus exécuter une itération. Cette perte d'énergie est traduite par les valences associées. Une explication plus détaillée mais ne rentrant pas dans les détails techniques, est fournie ci-dessous. Cette analyse a pour grands frères l'article de Simmons [Sim88] et celui de Leivant [Lei91].

Nous nous concentrerons sur les fonctions calculables en temps polynomial, et nous listons d'autres caractérisations de classe de complexité en fin de chapitre. Nous détaillerons et comparons l'approche de Bellantoni et Cook, et celle de Leivant. Nous avons apporté une nouvelle contribution qui est une caractérisation des classes DTIME($O(n^k)$).

4.1 Exponentielle comme un processus circulaire

Expliquons les grands traits de l'analyse prédicative de la récurrence. Une fonction définie par récurrence ou itération comporte une variable qui contrôle l'itération, i.e. un compteur, et le corps de l'itération qui est répété à chaque boucle. Supposons que tous deux sont de sorte $\mathbf{s}$. Le rôle attribué à chacun de ces objets est distingué par un type différent : la variable de contrôle est de type $\mathbf{S}(\mathbf{1})$ tandis que le type des résultats partiels et du résultat final est $\mathbf{S}(\mathbf{0})$. Ensuite, un objet de type $\mathbf{S}(\mathbf{0})$ ne peut être transformé en un objet de type $\mathbf{S}(\mathbf{1})$. Intuitivement une récurrence consomme une ressource apportée par le paramètre de récurrence, qui est de type $\mathbf{S}(\mathbf{1})$, et produit un résultat qui est sans énergie, i.e. de type $\mathbf{S}(\mathbf{0})$, qui ne peut plus servir de paramètre de récurrence. Cette stratification des données en $\mathbf{S}(\mathbf{0})$ et $\mathbf{S}(\mathbf{1})$, met en valeur une notion de ressources. Elle est formalisée par un système de types dans lequel

les arguments du prédicat **S** seront appelés des valences (*tier*).

Cette analyse permet de comprendre une définition exponentielle comme un processus circulaire. Regardons la définition suivante de la fonction exponentielle en base deux.

$$\exp(0) \to 1$$
$$\exp(n+1) \to \exp(n) + \exp(n)$$

L'addition + est une fonction définie par récurrence sur un de ses deux paramètres qui doit, d'après ce que nous avons dit, être de type $\mathbf{S}(\mathbf{1})$. Mais puisque la fonction exponentielle exp est définie par récurrence, son résultat est de type $\mathbf{S}(\mathbf{0})$. Il suit que $\exp(n)$ est aussi de type $\mathbf{S}(\mathbf{0})$, et cet argument n'a pas la bonne valence pour se connecter à + comme argument de récurrrence. Ainsi, cette définition n'est pas ramifiée car elle nécessite de recycler les données de type $\mathbf{S}(\mathbf{0})$ avec celles de type $\mathbf{S}(\mathbf{1})$.

4.2 L'approche de Leivant

4.2.1 Programmes pré-ramifiés

Nous reprenons la syntaxe et la sémantique des programmes de `E`, mais nous allons enrichir le système de types. A chaque sorte $\mathbf{s} \in \mathcal{S}$, nous associons un prédicat unaire **S**. Les arguments d'un prédicat, ainsi que leurs dénotations, sont appelés des *valences*. Les valences sont des termes bâtis à partir des symboles **0** et **suc**. Une valence représente un entier. Nous écrirons **k** à la place de $\mathbf{suc}^k(\mathbf{0})$. Les types sont définis à partir de l'ensemble $\mathcal{S}$ des sortes, par la grammaire suivante.

$$\begin{array}{llll} \textit{(Valences)} & \mathbf{k} & ::= & \mathbf{0} \mid \mathbf{suc}(\mathbf{k}) \\ \textit{(Atomes)} & \sigma & ::= & \mathbf{S}(\mathbf{k}) \qquad (\mathbf{s} \in \mathcal{S}) \\ \textit{(Types)} & \tau & ::= & \sigma \to \tau \mid \sigma \end{array}$$

Les règles de typage, voir figure 1.2, sont inchangées. Nous dirons qu'un terme τ est de valence k si t est de type $\mathbf{S}(\mathbf{k})$.

Définition 4.1. Un type τ est *pré-ramifié* ssi
- soit τ est un atome,
- soit $\tau = \mathbf{S}_1(\mathbf{k_1}), \ldots, \mathbf{S}_n(\mathbf{k_n}) \to \mathbf{S}(\mathbf{r})$ et $k_i \geq r$, pour tout $1 \leq i \leq n$.

Les valences des arguments sont toujours au moins aussi grandes que la valence **r** du résultat. Autrement dit, un calcul ne pourra pas accroître les valences d'un terme.

Définition 4.2. Un programme est pré-ramifié si le type de chaque symbole de fonction et de chaque constructeur est pré-ramifié[9].

Dans la suite, les types des symboles de fonction seront pré-ramifiés.

[9]En fait, il n'est pas nécessaire que le type d'un symbole de fonction soit pré-ramifié. Mais, cela simplifie la vie.

4.2.2 Itérations ramifiées

Nous sommes maintenant prêts à introduire la notion d'*itération ramifiée*. Cette notion est une des constructions fondamentales de ce mémoire.

Définition 4.3. Un type pré-ramifié τ est *ramifié* ssi

$$\tau = \mathbf{S(k)}, \mathbf{S}_1(\mathbf{v_1}) \ldots, \mathbf{S}_n(\mathbf{v_n}) \rightarrow \mathbf{S(r)} \qquad \text{et} \quad k > r$$

Autrement dit, la valence du premier paramètre doit être strictement supérieure à la valence du résultat.

$\mathbf{nat}_\omega$-itération Le symbole de fonction $\mathtt{f}$ est défini par $\mathbf{nat}_\omega$-itération si

$$\mathtt{f}(\mathbf{0}, \vec{y}) \rightarrow \mathtt{g}(\vec{y})$$
$$\mathtt{f}(\mathbf{suc}(x), \vec{y}) \rightarrow \mathtt{h}(\vec{y}, \mathtt{f}(x, \vec{y}))$$

où *le type de* $\mathtt{f}$ *est ramifié.*

Le *principe de ramification* de l'itération est que la valence $\mathbf{k}$ du paramètre de l'itération doit être strictement supérieure à la valence $\mathbf{r}$ du résultat.

$\mathbf{s}_\omega$-itération Le schéma d'itération ramifiée se généralise à chaque sorte $\mathbf{s}$. Le symbole de fonction $\mathtt{f}$ est défini par $\mathbf{s}_\omega$-itération si
pour chaque constante $\mathbf{b}:\mathbf{S(k)}$, il y a une équation

$$\mathtt{f}(\mathbf{b}, \vec{y}) \rightarrow \mathtt{g}_\mathbf{b}(\vec{y})$$

de même, pour chaque constructeur $\mathbf{c}:\mathbf{S}_1(\mathbf{r_1}), \ldots, \mathbf{S}_n(\mathbf{r_n}) \rightarrow \mathbf{S(k)}$,

$$\mathtt{f}(\mathbf{c}(x_1, \ldots, x_n), \vec{y}) \rightarrow \mathtt{h}_\mathbf{c}(\vec{y}, \mathtt{f}(x_{i_1}, \vec{y}), \ldots \mathtt{f}(x_{i_m}, \vec{y})) \qquad \text{où } x_{i_j}:\mathbf{S(k)}$$

où *le type de* $\mathtt{f}$ *est ramifié.*

4.2.3 Arithmétique ramifiée

Le principe de ramification crée des copies isomorphes de la structure **nat**. La valence $\mathbf{k}$ détermine la structure $\langle \mathbf{N(k)}, \mathbf{0_k}, \mathbf{suc_k} \rangle$ qui est une copie de $\langle \mathbf{nat}, \mathbf{0}, \mathbf{suc} \rangle$. Appelons *entier de valence* $\mathbf{k}$ un terme de la structure $\langle \mathbf{N(k)}, \mathbf{0_k}, \mathbf{suc_k} \rangle$. Un entier de valence $\mathbf{k}$ supporte les définitions par itérations d'entier de valence $< k$. Pour voir cela, commençons par l'addition et la multiplication.

$\mathtt{add}_k : \mathbf{N(k+1)}, \mathbf{N(k)} \rightarrow \mathbf{N(k)}$ et $[\![\mathtt{add}_k]\!](n, m) = n + m$

$$\mathtt{add}_k(\mathbf{0}_{k+1}, y) \rightarrow y$$
$$\mathtt{add}_k(\mathbf{suc}_{k+1}(x), y) \rightarrow \mathbf{suc}_k(\mathtt{add}_k(x, y))$$

$\mathtt{mul}_k : \mathbf{N(k+1)}, \mathbf{N(k+1)} \rightarrow \mathbf{N(k)}$ et $[\![\mathtt{mul}]\!](n, m) = n \cdot m$

$$\mathtt{mul}_k(\mathbf{0}_{k+1}, y) \rightarrow \mathbf{0}_k$$
$$\mathtt{mul}_k(\mathbf{suc}_{k+1}(x), y) \rightarrow \mathtt{add}_k(y, \mathtt{mul}_k(x, y))$$

Ensuite, nous pouvons définir un opérateur de coercition qui transforme un entier de valence $k+1$ en un entier de k. L'inverse est impossible.

$\mathtt{coerce}_k : \mathbf{N}(\mathbf{k}+\mathbf{1}) \to \mathbf{N}(\mathbf{k})$ et $[\![\mathtt{coerce}_k]\!](n) = n$

$$\begin{aligned} \mathtt{coerce}_k(\mathbf{0}_{k+1}) &\to \mathbf{0}_k \\ \mathtt{coerce}_k(\mathbf{suc}_{k+1}(x)) &\to \mathbf{suc}_k(\mathtt{coerce}_k(x)) \end{aligned}$$

Nous pouvons composer des programmes définis par itération.

$\mathtt{cube}_k : \mathbf{N}(\mathbf{k}+\mathbf{2}) \to \mathbf{N}(\mathbf{k})$ et $[\![\mathtt{cube}_k]\!](n) = n^3$

$$\mathtt{cube}_k(x) \to \mathtt{mul}_k(\mathtt{coerce}_{k+1}(x), \mathtt{mul}_{k+1}(x,x))$$

Le coût de la composition est un accroissement des valences, ce qui nécessite l'utilisation de deux copies du programme `mul`.

Remarque 4.4. La différence entre les entiers de valence 1 et de valence 0 s'exprime en λ-calcul, comme l'on démontré Leivant et Marion [LM93a, LM93b], par deux représentations distinctes. Les entiers de valence 1 sont représentés par des entiers de Church de la forme $\lambda f \lambda x.f^n(x)$, et les entiers de valence 0 sont des termes de la structure **nat**.

4.2.4 Une première caractérisation de PTIME

Définition 4.5. La classe $\mathbf{SPR}^\omega$ est la classe des programmes pré-ramifiés où chaque symbole de fonction est défini soit explicitement, soit par itération ramifiée.

D. Leivant a démontré dans [Lei94c] le résultat suivant.

Théorème 4.6. *La classe de fonctions* $\mathcal{F}(\mathbf{SPR}^\omega)$ *est exactement la classe* PTIME *des fonctions calculables en temps polynomial.*

En fait, dans le même article, Leivant a observé qu'il suffit de ne considérer que les valences **0** et **1** pour caractériser PTIME.

Corollaire 4.7. *Soit* $\mathbf{SPR}^\omega(0,1)$ *la classe des programmes dont chaque terme est de valence* 0 *ou* 1. *La classe de fonctions* $\mathcal{F}(\mathbf{SPR}^\omega(0,1))$ *est exactement la classe* PTIME *des fonctions calculables en temps polynomial.*

Exemple 4.8. Une itération de longueur polynomiale est obtenue en imbriquant des itérations. En partant de $\mathtt{f}_0 : \mathbf{N}(\mathbf{0}) \to \mathbf{N}(\mathbf{0})$, nous construisons la suite $(\mathtt{f}_{k+1} : \mathbf{N}(\mathbf{1})^{k+1}, \mathbf{N}(\mathbf{0}) \to \mathbf{N}(\mathbf{0}))$ comme suit :

$$\begin{aligned} \mathtt{f}_{k+1}(\mathbf{0}, \vec{x}, y) &\to y \\ \mathtt{f}_{k+1}(\mathbf{suc}(t), \vec{x}, y) &\to \mathtt{f}_k(\vec{x}, \mathtt{f}_{k+1}(t, \vec{x}, y)) \end{aligned}$$

Nous avons $[\![\mathtt{f}_{k+1}]\!](m_1, \ldots, m_{k+1}, p) = [\![\mathtt{g}]\!]^{\prod m_i}(p)$ Leivant [Lei94c] a établi que $\mathrm{DTIME}(O(n^k))$ est caractérisée exactement par le degré d'imbriquation des itérations.

4.3 Itérations strictement ramifiées

Définition 4.9. Un type τ est *k-ramifié* ssi

$$\tau = \mathbf{S(k)}, \mathbf{S}_1(\mathbf{v_1}), \ldots, \mathbf{S}_n(\mathbf{v_n}) \to \mathbf{S}_{n+1}(\mathbf{0}) \quad \text{avec } k > v_i \geq 0, \text{ pour } 1 \leq i \leq n$$

Un type est strictement ramifié s'il est k-ramifié, pour un certain k.

Le type $\mathbf{N(k+1)}, \mathbf{N(k)}, \mathbf{N(k)} \to \mathbf{N(0)}$ est $k+1$-ramifié. Par contre, le type $\mathbf{N(k)}, \mathbf{N(k)} \to \mathbf{N(0)}$ ne l'est pas. Les types des exemples $\mathtt{add}_k$ et $\mathtt{mul}_k$ ne sont pas strictement ramifiés si $k > 0$. Cependant, nous pouvons assigner à `add` le type $\mathbf{N(1)}, \mathbf{N(0)} \to \mathbf{N(0)}$, et à `mul` le type $\mathbf{N(2)}, \mathbf{N(1)} \to \mathbf{N(0)}$.

$\mathbf{s}_k$-itération Le schéma *$\mathbf{s}_k$-itération* est le suivant.

Pour chaque constante $\mathbf{b} : \mathbf{S(k)}$, il y a une équation

$$\mathtt{f}(\mathbf{b}, \vec{y}) \to \mathtt{g}_{\mathbf{b}}(\vec{y})$$

de même, pour chaque constructeur $\mathbf{c} : \mathbf{S}_1(\mathbf{r_1}), \ldots, \mathbf{S}_n(\mathbf{r_n}) \to \mathbf{S(k)}$,

$$\mathtt{f}(\mathbf{c}(x_1, \ldots, x_n), \vec{y}) \to \mathtt{h}_{\mathbf{c}}(\vec{y}, \mathtt{f}(x_{i_1}, \vec{y}), \ldots \mathtt{f}(x_{i_m}, \vec{y})) \qquad \text{où } x_{i_j} : \mathbf{S(k)}$$

où *le type de* `f` *est k-ramifié.*

Techniquement, les valences des arguments de $\mathtt{h}_{\mathbf{c}}$ sont $< k$. Ensuite, à chaque imbrication d'un schéma d'itération, la valence du paramètre d'itération augmente. Il n'est plus possible de faire une itération "par composition" comme celle de `cube`, car la valence du résulat est bloquée à 0.

Définition 4.10. La classe $\mathbf{SPR}^k$ est la classe des programmes pré-ramifiés où chaque symbole de fonction est soit défini explicitement, soit par $\mathbf{s}_r$-itération avec $r \leq k$.

Il est pratique d'ajouter la règle de typage de la figure 4.1. La règle de coercition exprime qu'une donnée de valence $k+1$ peut être réduite à k. Cette régle n'est pas nécessaire, car le programme $\mathtt{coerce}_k$ exerce, déjà, une telle coercition.

$$\frac{\Gamma, \Theta \vdash t : \mathbf{N(suc(k))}}{\Gamma, \Theta \vdash t : \mathbf{N(k)}}\ \mathbf{suc}E$$

FIG. 4.1 – Coercition.

Exemple 4.11. Soit $\mathtt{g} : \mathbf{N(0)} \to \mathbf{N(0)}$.

$\mathtt{F}_0 : \mathbf{N(1)}, \mathbf{N(0)}, \mathbf{N(0)} \to \mathbf{N(0)}$ et $[\![\mathtt{F}_0]\!](m, n, p) = [\![\mathtt{g}]\!](p)$

$$\mathtt{F}_0(t, x, y) \to \mathtt{g}(y)$$

$\mathtt{F}_{k+1} : \mathbf{N}(\mathbf{k+1}), \mathbf{N}(\mathbf{k}), \mathbf{N}(\mathbf{0}) \to \mathbf{N}(\mathbf{0})$ et $[\![\mathtt{F}_{k+1}]\!](m,n,p) = [\![\mathtt{g}]\!]^{m.n^k}(p)$

$$\begin{aligned} \mathtt{F}_{k+1}(\mathbf{0}, x, y) &\to y \\ \mathtt{F}_{k+1}(\mathbf{suc}(t), x, y) &\to \mathtt{F}_k(x, x, \mathtt{F}_{k+1}(t, x, y)) \end{aligned}$$

Dans le typage de la deuxième équation, nous utilisons la règle de coercition.

Théorème 4.12. *La classe de fonctions $\mathcal{F}(\mathbf{SPR}^k)$ est exactement la classe* DTIME$(O(n^k))$ *des fonctions calculables en temps $O(n^k)$. Par conséquent,* $\cup_k \mathcal{F}(\mathbf{SPR}^k) =$ PTIME.

L'idée de la ramification stricte provient des travaux de Bloch [Blo94] et de Leivant et Marion [LM00a]. Un résultat similaire a été obtenu par Covino, Pani et Caporaso [CPC00b] en méta-stratifiant le schéma d'itération sûre de Bellantoni et Cook.

4.4 L'approche de Bellantoni et Cook

Dans l'article [BC92], Bellantoni et Cook ont défini une algèbre de fonctions B qui est la classe PTIME. Le mécanisme de limitation du calcul est similaire à celui que nous venons de décrire. Les fonctions ont deux types d'arguments nommés normaux (*normal*) et sûrs (*safe*). Les paramètres de récurrence sont toujours des arguments normaux. Le résultat d'une récurrence est un argument sûr. Enfin, le schéma de composition sûre ne permet pas à un argument sûr de muter en argument normal. Dans une première approximation, nous pouvons identifier un argument normal avec un argument de valence 1 et un argument sûr comme un argument de valence 0.

Nous présentons le langage de programmation `B` qui calcule les fonctions de l'algèbre B. L'ensemble des données est **word**. Les programmes de `B` sont les programmes définis uniquement grâce aux schémas ci-dessous. En suivant la convention de [BC92], les arguments normaux $\vec{x}$ sont écrits à gauche, et les arguments sûrs $\vec{y}$ à droite. Ils sont séparés par un point-virgule, *e.g.* $\mathtt{f}(\vec{x}; \vec{y})$.

Définition explicite sûre

$$\mathtt{f}(p_1, \ldots; p_n, \ldots) \to t \qquad\qquad t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$$

Composition sûre

$$\mathtt{f}(\vec{x}; \vec{y}) \to \mathtt{g}(\mathtt{h}_1(\vec{x};); \mathtt{h}_2(\vec{x}; \vec{y}))$$

Récursion sûre

$$\begin{aligned} \mathtt{f}(\boldsymbol{\epsilon}, \vec{x}; \vec{y}) &\to \mathtt{g}(\vec{x}; \vec{y}) \\ \mathtt{f}(\mathbf{i}(z), \vec{x}; \vec{y}) &\to \mathtt{h}_i(z, \vec{x}; \vec{y}, \mathtt{f}(z, \vec{x}; \vec{y})) \end{aligned}$$

Le programme `f` de l'exemple 4.8 se traduit en séparant les arguments ainsi $\mathtt{f}(t, \vec{x}; y)$. La différence avec l'approche de Leivant réside dans la composition

qui est plus libre car le résultat n'est pas typé. Par exemple, le programme `cube` n'a pas besoin de plusieurs copies de `mul` à des valences différentes.

$$\begin{aligned}
\mathtt{add}(\mathbf{0}; y) &\to y \\
\mathtt{add}(\mathbf{suc}(x); y) &\to \mathbf{suc}(\mathtt{add}(x; y)) \\
\mathtt{mul}(\mathbf{0}, y;) &\to \mathbf{0} \\
\mathtt{mul}(\mathbf{suc}(x), y;) &\to \mathtt{add}(y; \mathtt{mul}(x, y;)) \\
\mathtt{cube}(x;) &\to \mathtt{mul}(x, \mathtt{mul}(x, x;);)
\end{aligned}$$

Théorème 4.13. *La classe de fonctions $\mathcal{F}(\mathtt{B})$ est exactement la classe* PTIME *des fonctions calculables en temps polynomial.*

4.5 Comparaison avec les autres approches

L'étude des classes de fonctions de faible complexité algorithmique remonte aux travaux de Cobham [Cob62] et Ritchie [Rit63]. La démarche consiste à définir une algèbre de fonctions engendrée par des fonctions initiales, close par composition et par des schémas de récursion bornée. Cette approche s'inscrit dans le prolongement des travaux sur les hiérachies de fonctions, comme par exemple ceux de Kalmar, Grzegorczyk ou de Péter. Nous reparlerons de cela dans le prochain chapitre.

Cobham a caractérisé PTIME en définissant l'algèbre C de fonctions sur les entiers qui contient les fonctions initiales :

1. la fonction nulle,
2. les projections,
3. les fonctions successeurs $s_0(x) = 2x$ et $s_1(x) = 2x + 1$,
4. la fonction *smash* $x\#y = 2^{|x|\cdot|y|}$ où $|x| = \lceil \log_2(x+1) \rceil$.

et qui est close par composition et par récursion bornée en notation.

$$\begin{aligned}
f(0, \vec{x}) &= g(\vec{x}) \\
f(s_0(t), \vec{x}) &= h(t, \vec{x}, f(t, \vec{x})) \\
f(s_1(t), \vec{x}) &= h(t, \vec{x}, f(t, \vec{x})) \\
& \qquad f(t, x) \leq k(t, \vec{x}) \quad k \text{ est une fonction de } C
\end{aligned}$$

Ce type de modélisation est critiqué à cause de l'aspect incestueux du schéma de récurrence bornée. La fonction f n'est définie que si elle est bornée par une fonction k qui est, déjà, dans la classe C. La nature de cette construction nécessite d'amorcer la construction avec des fonctions initiales ad-hoc. Un résumé des différentes caractérisations ainsi obtenues, se trouve dans l'article de Clote [Clo95].

L'autre mouvement, dont nous avons parlé dans le chapitre 3, considère des schémas de programme interprétés sur des modèles finis. Les fonctions étudiées sont représentées par un sytème formel, ce qui répond à la critique précédente. Cependant, le domaine de calcul est fini. Ce parti pris est justifiable car les ordinateurs ont des ressources de calcul limitées, comme la mémoire. D'un autre côté, la théorie de la programmation ne repose pas sur cette intuition.

La méthodologie de l'analyse prédicative est plus satisfaisante car elle permet, dans un premier temps, de modéliser les fonctions de faibles complexité par un système formel minimal interprété sur des domaines potentiellement infinis. La différence avec les deux approches citées repose sur le fait que l'analyse prédicative se concentre sur la structure du calcul, plutôt que sur une limitation externe des ressources de calcul. Ainsi, dans un second temps, il nous paraît possible d'étudier, non plus les fonctions de faible complexité, mais les algorithmes de faible complexité. Pour y parvenir, il faut élargir la classe des algorithmes capturés. Ce sujet sera abordé dans la partie suivante.

4.6 Autres classes de complexité

Le fil conducteur a été la classe PTIME et ses caractérisations. L'analyse prédicative a permis de modéliser des classes de complexité importantes. Nous avons classé ces résulats suivant l'approche utilisée.

L'approche de Bellantoni et Cook

- Bellantoni [Bel94] a caractérisé les langages NPTIME et la hiérarchie polynomiale.
- Bloch [Blo94] a caractérisé les fonctions de NC_1 = ALOGTIME et de POLYLOGSPACE.

L'approche de Leivant

- Leivant et Marion [LM95, LM97, LM00b] ont caractérisé les fonctions de PSPACE.
- Leivant et Marion [LM00a] ont caractérisé les fonctions NC_1 = ALOGTIME.
- Leivant [Lei98] a caractérisé les fonctions de NC.

Chapitre 5

Au-dessus de l'exponentielle

Ce chapitre est une digression pour expliquer comment les différentes approches de l'analyse prédicative s'insèrent dans la hiérarchie de Grzegorczyk. Bien que les fonctions auront une complexité démesurée, il nous semble ces études permettent de mieux comprendre l'analyse prédicative et son rapport aux fondements des mathématiques.

Dans un premier temps, nous partirons de la construction d'Ackermann pour définir une fonction qui n'est pas primitive récursive. Nous verrons comment diagonaliser les définitions par récurrence ramifiée soit de façon externe en suivant les travaux de Caporaso, Pani et Covino soit de façon interne comme nous le proposons.

Ensuite, nous expliquerons comment Leivant a ramifié le système T de Gödel et la relation avec les itérateurs de Church du λ-calcul simplement typé.

Nous terminerons avec la définition de mesure fine du degré de récurrence proposée par Bellantoni et Niggl, dans la tradition de Heinermann [Hei61], Schwichtenberg [Sch69] et Müller [Mül74].

5.1 Par diagonalisation

De l'exposé des sections précédentes, nous pouvons dégager trois critères de notre démarche : (i) les fonctions sont définies par des schémas d'équations, (ii) leurs définitions suivent un principe de récurrence, et enfin (iii) la récurrence obéit au principe de ramification. Bien entendu, il existe des fonctions dont la définition vérifie les trois points mentionnés et qui ne sont pas à croissance polynomiale. Il suffit de considérer un schéma de récursion imbriquée.

$\mathtt{f} : \mathbf{N}(1), \mathbf{N}(0) \to \mathbf{N}(0)$

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

Le programme $\mathtt{f}$ calcule la fonction exponentielle, plus précisément, nous avons $[\![\mathtt{f}]\!](n, m) = 2^n + m$.

5.1.1 La fonction d'Ackermann

En 1928, Ackermann [Ack28] construit une fonction qui n'est pas primitive récursive. Les équations ci-dessous suivent la définition de Ritchie [Rit65].

$$\begin{aligned}
\mathtt{A}_0(x,y) &\to \mathbf{suc}(y) \\
\mathtt{A}_1(x,y) &\to \mathtt{add}(x,y) \\
\mathtt{A}_2(x,y) &\to \mathtt{mul}(x,y) \\
\mathtt{A}_{k+1}(\mathbf{0},y) &\to \mathbf{suc}(\mathbf{0}) && (k \geq 2) \\
\mathtt{A}_{k+1}(\mathbf{suc}(x),y) &\to \mathtt{A}_k(\mathtt{A}_{k+1}(x,y),y)
\end{aligned}$$

La suite de fonctions $(\llbracket \mathtt{A}_k \rrbracket)_{k\in\mathbb{N}}$ est croissante. Individuellement, chaque fonction $\llbracket \mathtt{A}_k \rrbracket$ est primitive récursive. La hiérarchie de Grzegorczyk [Grz53] est engendrée par la suite $(\llbracket \mathtt{A}_k \rrbracket)_{k\in\mathbb{N}}$. La classe $\mathcal{G}(k)$ est la plus petite classe de fonction qui contient le successeur, la fonction nulle, les projections, la fonction $\llbracket \mathtt{A}_k \rrbracket$ et qui est close par composition et récursion bornée. L'inclusion de $\mathcal{G}(k)$ dans $\mathcal{G}(k+1)$ est stricte car la fonction $\lambda x.\llbracket \mathtt{A}_{k+1} \rrbracket(x,x)$ majore toutes les fonctions de $\mathcal{G}(k)$ et appartient à $\mathcal{G}(k+1)$. D'autre part, $\cup_{n\in\mathbb{N}}\mathcal{G}(n)$ forme la classe des fonctions primitives récursives.

En transformant le paramètre k en argument, nous obtenons un programme défini par *récursion multiple*, dont nous ne reportons que les deux dernières équations.

$$\begin{aligned}
\mathtt{A}(\mathbf{suc}(k),\mathbf{0},y) &\to \mathbf{suc}(\mathbf{0}) \\
\mathtt{A}(\mathbf{suc}(k),\mathbf{suc}(x),y) &\to \mathtt{A}(k,\mathtt{A}(\mathbf{suc}(k),x,y),y)
\end{aligned}$$

Ackermann obtient, par diagonalisation, une fonction $\lambda n.\llbracket \mathtt{A} \rrbracket(n,n,n)$ qui n'est pas primitive récursive, car elle majore toutes les fonctions primitives récursives. Simmons, dans le livre [Sim00], décortique la construction de la fonction d'Ackermann.

5.1.2 Diagonalisation avec types dépendants ramifiés

L'exemple 4.11 définit une suite de fonctions $(F_k)_{k\in\mathbb{N}}$. Le passage de F_k à F_{k+1} fait sauter de la classe DTIME($O(n^k)$) à DTIME($O(n^{k+1})$). En diagonalisant par le procédé d'Ackermann, nous obtenons un programme $\mathtt{F}$ défini par double récursion.

$$\begin{aligned}
\mathtt{F}(\mathbf{0},t,x,y) &\to \mathtt{g}(y) && \mathtt{g} : \mathbf{N}(\mathbf{0}) \to \mathbf{N}(\mathbf{0}) \\
\mathtt{F}(\mathbf{suc}(k),\mathbf{0},x,y) &\to y \\
\mathtt{F}(\mathbf{suc}(k),\mathbf{suc}(t),x,y) &\to \mathtt{F}(k,x,x,\mathtt{F}(\mathbf{suc}(k),t,x,y))
\end{aligned}$$

La fonction $\llbracket F \rrbracket(k+1,m,n,p)$ calcule $g^{m\cdot n^k}(p)$, et donc la fonction exponentielle car en posant $g = \mathbf{suc}$, nous avons $\llbracket F \rrbracket(k,2,2,0) = 2^k$.

Le principe de ramification s'étend à ce schéma de définition par double récursion. Pour cela, il nous faut ajouter au système de types le quantificateur Π afin d'introduire des types dépendants des termes. Nous nous référons au livre de Martin-Löf [ML84], sur ce sujet.

Avant tout, il nous semble plus intéressant de comprendre le mécanisme. Nous pouvons assigner à `F` le type

$$\Pi\alpha.\mathbf{N}(\mathbf{suc}(\alpha)), \mathbf{N}(\alpha), \mathbf{N}(\mathbf{0}) \rightarrow \mathbf{N}(\mathbf{0})$$

Le type de $\mathtt{F}(k)$ est $\mathbf{N}(\mathbf{suc}(k)), \mathbf{N}(k), \mathbf{N}(\mathbf{0}) \rightarrow \mathbf{N}(\mathbf{0})$ et il est identique au type de $\mathtt{F}_k$.

Formellement, les valences sont enrichies de la constante ω.

(Variables)		::=	$\alpha \mid \ldots$
(Valences)	$\mathbf{k}$	::=	$\mathbf{0} \mid \mathbf{suc}(\mathbf{k}) \mid \omega \mid \alpha$
(Atomes)	σ	::=	$\mathbf{S}(u)$
(Propositions)	τ	::=	$\sigma \rightarrow \tau \mid \sigma$
(Types)	ρ	::=	$\Pi\alpha.\tau \mid \tau$

Nous ajoutons aux règles de typage des figures 1.2 et 4.1 celle de la figure 5.1.2

$$\frac{}{\vdash \mathbf{0} : \mathbf{N}(\omega)}$$

$$\frac{}{\vdash \mathbf{suc} : \mathbf{N}(\omega) \rightarrow \mathbf{N}(\omega)}$$

$$\frac{\Theta \vdash k : \mathbf{N}(\omega) \quad \Gamma, \Theta \vdash t : \Pi\alpha.\tau}{\Gamma, \Theta \vdash t(k) : \tau[\alpha \leftarrow k]}\ \Pi E$$

FIG. 5.1 – Produit dépendant.

Pour aller plus loin, il faut traduire cette construction en λ-calcul à cause du système de type qui est délicat. Nous n'avons fait qu'esquisser une construction dont l'exploration semble intéressante. L'une des raisons est que nous montrons clairement comment un processus exponentiel résulte d'une diagonalisation.

5.1.3 L'exponentielle par diagonalisation externe

Regardons du côté des hiérarchies de fonctions et tout particulièrement de la hiérarchie à croissance lente, voir Cichon et Wainer [CW83]. Celle-ci est une suite de fonctions indexées par des ordinaux $< \epsilon_0$:

$$\begin{aligned} g_0(x) &= 0 \\ g_{\alpha+1}(x) &= g(g_\alpha(x)) \\ g_\mu(x) &= g_{\mu[x]}(x) \qquad (\mu \text{ est un ordinal limite}) \end{aligned}$$

où $\mu[]$ est une suite fondamentale qui est pré-définie telle que μ est la borne supérieure de la suite $\mu[0], \mu[1], \ldots$. Dans le cas de la hiérarchie à croissance

lente, les choses se simplifient : $g_{\mu[x]}$ est obtenu en remplaçant ω par x dans la forme normale de Cantor en base ω de μ. Par exemple, $g_{(\omega^\omega+\omega)[x]} = g_{x^x+x}$. Le passage à la limite est effectué en diagonalisant la suite de fonctions $(g_n)_{n\in\mathbb{N}}$.

Dans le même esprit, Caporaso, Pani et Covino [CPC00a] ont proposé une fonction `cdiag` pour diagonaliser la suite $(\mathtt{F}_k)_{k\in\mathbb{N}}$.

$$\mathtt{cdiag}(k,x,y) \to \mathtt{F}_k(k,x,y)$$

Cette approche permet de capturer la hiérarchie des fonctions élémentaires, ainsi que la hiérarchie de Grzegorczyk. La différence avec la construction précédente est méthodologique. La diagonalisation est faite "au-dessus" d'une classe de fonctions.

5.2 Ramification du système T

Le système T de Gödel [Göd58] généralise la récursion primitive aux fonctionnelles. Au premier rang, nous retrouvons les fonctions primitives récursives. Au deuxième rang, quand un programme peut itérer une fonction, la fonction d'Ackermann est définissable, et ainsi de suite.

Leivant [Lei99b] a considéré les valences comme des types. Les valences sont ordonnées suivant le rang du type. Concrètement, un constructeur de type Ω est ajouté. Le type $\Omega(\tau)$ est le type qui supporte une définition par récurrence d'une construction de type τ. Le système de type est réduit à :

$$\textit{(Type)} \quad \tau \quad ::= \quad \sigma \to \tau \mid \Omega(\tau) \mid \mathbf{s}$$

Le type $\mathbf{s}$ est le seul type atomique. Le type $\Omega(\tau)$ est un type de rang 0 dont l'interprétation est identique au type atomique $\mathbf{s}$. Illustrons cette idée par des exemples et commençons par une itération simple.

$$\begin{aligned} \mathtt{add}(\mathbf{0}, y) &\to y \\ \mathtt{add}(\mathbf{suc}(x), y) &\to \mathbf{suc}(\mathtt{add}(x, y)) \end{aligned}$$

Le type de `add` est $\Omega(\mathbf{nat}), \mathbf{nat} \to \mathbf{nat}$. Intuitivement, $\Omega(\mathbf{nat})$ correspond à $\mathbf{N}(1)$ et $\mathbf{nat}$ à $\mathbf{N}(\mathbf{0})$. Ainsi, les programmes dont les types n'utilisent que $\Omega(\mathbf{s})$ sont calculables en temps polynomial, et réciproquement. Nous retrouvons le théorème 4.7.

L'exponentielle est définie par une itération à l'ordre supérieur :

$$\begin{aligned} \mathtt{D} &\to \lambda X^{\tau\to\tau} \lambda y^{\tau}. X(Xy) \\ \mathtt{E}_1(0) &\to \mathtt{g} \qquad\qquad \mathtt{g} : \tau \to \tau \\ \mathtt{E}_1(\mathbf{suc}(n)) &\to \mathtt{D}(\mathtt{E}_1(n)) \end{aligned}$$

Nous obtenons $[\![\mathtt{E}_1]\!](n) = [\![\mathtt{g}]\!]^{2^n}$ et le type de $\mathtt{E}_1$ est $\Omega(\tau \to \tau) \to (\tau \to \tau)$. Par composition, nous avons la fonction exponentielle itérée.

$$\mathtt{E}_{k+1}(x) \to \mathtt{E}_1(\mathtt{E}_k(x))$$

Nous avons $[\![\mathtt{E}_k]\!](n) = [\![\mathtt{g}]\!]^{\exp_k(n)}$ et nous pouvons vérifier que le type de $\mathtt{E}_k$ est $\Omega^k(\mathbf{nat}) \to \mathbf{nat}$.

Leivant a appliqué le principe de ramification au système T. Il a ainsi construit une hiérarchie de fonctions qui débute avec les fonctions de PTIME suivies des fonctions de DTIME($\exp_k(n)$). Les fonctions de DTIME($\exp_k(n)$) sont dites élémentaires, au sens de Kalmar, et correspondent à la classe $\mathcal{G}(3)$ de la hiérarchie de Grzegorczyk. Une autre caractérisation des fonctions élémentaires est obtenue en considérant la hiérarchie à croissance lente en dessous de ϵ_0.

Dans le même article, Leivant propose un autre éclairage. La ramification du système T s'exprime comme une extension de la caractérisation de Leivant et Marion [LM93a] de PTIME en λ-calcul simplement typé auquel nous avons ajouté des constructeurs et des destructeurs de sorte. Bohm et Berarducci [BB85] ont généralisé la notion d'entier de Church à toutes les structures de données. Une définition par récurrence à partir d'un paramètre de type $\Omega(\tau)$ se traduit par un itérateur de Church de type $(\tau \to \tau) \to (\tau \to \tau)$, et réciproquement. Or, d'après Fortune, Leivant et O'Donnell [FLO83], le taux de croissance des fonctions définies en λ-calcul simplement typé est au plus élémentaire. En conclusion, nous retrouvons la caractérisation sus-citée des fonctions calculées en temps borné par $\exp_k$.

5.3 Mesure sur la récursion primitive

Niggl [Nig98, Nig00] et Bellantoni et Niggl [BN99] ont caractérisé la hiérarchie de Grzegorczyk en commençant avec la classe PTIME. Pour cela, ils ont repris le procédé de séparation des arguments empruntés à la caractérisation de PTIME de Bellantoni et Cook [BC92]. Une fonction ρ mesure le rang des arguments d'un symbole de fonction. Grossièrement, le rang d'un argument est le nombre de fois qu'il sert comme paramètre de récursion. Nous invitons le lecteur à consulter l'un ou l'autre des articles pour une définition exacte. Nous nous contenterons d'illustrer la fonction de rang ρ de [BN99].

$$\begin{aligned} \mathtt{add}(\mathbf{0}, y) &\to y \\ \mathtt{add}(\mathbf{suc}(x), y) &\to \mathbf{suc}(\mathtt{add}(x, y)) \end{aligned}$$

Le rang du premier argument est $\rho(\mathtt{add}, 1) = 1$ car cet argument est le paramètre d'une récurrence. Au contraire, $\rho(\mathtt{add}, 2) = 0$ car le deuxième argument n'est pas un paramètre de récurrence dans le programme `add`.

$$\begin{aligned} \mathtt{mul}(\mathbf{0}, y) &\to \mathbf{0} \\ \mathtt{mul}(\mathbf{suc}(x), y) &\to \mathtt{add}(y, \mathtt{mul}(x, y)) \end{aligned}$$

Nous avons $\rho(\mathtt{mul}, 1) = 1$ car le premier argument est le paramètre d'une récurrence dans laquelle l'argument critique n'est pas utilisé comme paramètre de récurrence, puisque $\rho(\mathtt{add}, 2) = 0$. Maintenant, $\rho(\mathtt{mul}, 2) = 1$ car le deuxième argument est utilisé comme paramètre de récurrence dans le programme `add`, puisque $\rho(\mathtt{add}, 1) = 1$.

$$\begin{aligned} \mathtt{exp}(\mathbf{0}) &\to \mathbf{0} \\ \mathtt{exp}(\mathbf{suc}(x)) &\to \mathtt{add}(\mathtt{exp}(x), \mathtt{exp}(x)) \end{aligned}$$

Ici, $\rho(\exp, 1) = \rho(\texttt{add}, 1) + 1 = 2$ car l'argument de exp est un paramètre d'une récurrence dans laquelle l'argument critique $\exp(x)$ sert de paramètre de récurrence dans le programme `add`.

Remarque 5.1. Nous avons connaissance de la caractérisation des fonctions élémentaires par Oitavem [Oit97] et par Beckmann et Weiermann [BW96].

Chapitre 6

Synthèse de programmes efficaces en théories des types

Une théorie logique H définit un ensemble, récursivement énumérable, de fonctions totales. Ainsi, l'arithmétique de Peano définit les fonctions programmées par le système T (Gödel [Göd58]), et l'arithmétique du second ordre définit les fonctions programmées par le système F (Girard [Gir72]). Existe-t-il une théorie logique des fonctions calculables en temps polynomial ? Buss [Bus86], Leivant [Lei91] et d'autres ont construit de tels systèmes logiques. Dans ce chapitre, nous proposons une restriction de l'arithmétique de Peano, qui caractérise les fonctions calculables en temps polynomial. Cette restriction porte sur la quantification universelle qui est limitée à une catégorie de termes, appelée canonique. Le formalisme s'inspire de l'arithmétique fonctionnelle du second ordre AF_2 de Leivant [Lei83]. Ce résultat est une contribution originale de ce mémoire. En contrepartie, ce chapitre est plus ardu.

Une motivation est la question, un peu floue, d'appréhender ce que pourrait être les mathématiques attachées à la réalité du temps polynomial, comme il y a les mathématiques constructives. Plus concrètement, la théorie des types de Martin-Löf [ML84], le calcul des constructions [CH88] sont des théories logiques à partir desquelles il est possible de synthétiser des programmes sûrs à partir d'une démonstration de leurs corrections. Dans cette perspective, notre apport est la possibilité de certifier la complexité du programme synthétisé.

6.1 Arithmétique à quantification ramifiée

6.1.1 Logique du premier ordre à quantification ramifiée

Nous présentons le système **ST** dont la caractéristique principale est la restriction du domaine de la quantification universelle à une certaine classe de termes, dits canoniques, en suivant la terminologie du livre [ML84] de Martin-Löf. Un terme canonique est un terme bâti uniquement à partir des constructeurs et des variables. Les différentes catégories de termes sont résumées ci-

dessous.

$$
\begin{array}{llll}
\textit{(Constructeurs)} & \mathcal{T}(\mathcal{C}) \ni \mathbf{a} & ::= & \mathbf{c}(\mathbf{a}_1)\dots(\mathbf{a}_n) \mid \mathbf{b} \\
\textit{(Termes)} & \mathcal{T}(\mathcal{C},\mathcal{F},\mathcal{X}) \ni t & ::= & x \mid \mathbf{c}(t_1)\dots(t_n) \mid \mathtt{f}(t_1)\dots(t_n) \\
\textit{(Termes clos)} & \mathcal{T}(\mathcal{C},\mathcal{F}) \ni s & ::= & \mathbf{c}(s_1)\dots(s_n) \mid \mathtt{f}(s_1)\dots(s_n) \\
\textit{(Canoniques)} & \mathcal{T}(\mathcal{C},\mathcal{X}) \ni p & ::= & \mathbf{c}(p_1)\dots(p_n) \mid x \\
 & & & \mathbf{c},\mathbf{b} \in \mathcal{C}, \mathtt{f} \in \mathcal{F} \text{ et } x \in \mathcal{X}
\end{array}
$$

Le système **ST** est un calcul en déduction naturelle dont les règles sont écrites dans la figure 6.1. **ST** diffère du fragment $(\rightarrow, \wedge, \forall)$ de la logique minimale par le traitement de la quantification. L'élimination du quantificateur universel est restreinte aux termes canoniques :

$$\frac{\forall x.A}{A[x \leftarrow p]}\,\forall E \text{ où } p \in \mathcal{T}(\mathcal{C},\mathcal{X})$$

Nous nous reportons au texte classique [Pra65] de Prawitz ou au livre [GLT89] de Girard, Lafont et Taylor, pour une introduction à la déduction naturelle.

Remarque 6.1. Les termes canoniques avaient été baptisés motifs, au chapitre 1. Ce changement de désignation se justifie par l'importance de cette classe de termes, dans ce chapitre.

Prémisses : $x : A$ Un prédicat A étiqueté par x

Règles d'introduction	*Règles d'élimination*
$\dfrac{\begin{array}{c}\{x:A\}\\ \vdots \\ B\end{array}}{A \rightarrow B} \rightarrow I$	$\dfrac{A \rightarrow B \qquad A}{B} \rightarrow E$
$\dfrac{A_1 \qquad A_2}{A_1 \wedge A_2} \wedge I$	$\dfrac{A_1 \wedge A_2}{A_j} \wedge E$
$\dfrac{A[y]}{\forall x.A} \forall I$	$\dfrac{\forall x.A}{A[x \leftarrow p]} \forall E, p \in \mathcal{T}(\mathcal{C},\mathcal{X})$

Restriction d'application des règles dans une dérivation :
– Dans la règle $(\forall I)$, y ne doit apparaître ni dans les prémisses, ni dans la conclusion $A[x]$.
• *Dans la règle* $(\forall E)$*, le terme* p *est canonique,* i.e. $p \in \mathcal{T}(\mathcal{C},\mathcal{X})$.

FIG. 6.1 – Règles logiques de **ST**.

Nous noterons $A[t]$ pour indiquer que le terme t peut apparaître dans la formule A. Par contraste, nous écrirons $\mathbf{N}(t)$ quand le terme t est l'argument du prédicat unaire $\mathbf{N}$.

$$\frac{}{\mathbf{N}(\mathbf{0})}\,\mathbf{0}I \qquad \frac{\mathbf{N}(t)}{\mathbf{N}(\mathbf{suc}(t))}\,\mathbf{suc}I$$

$$\frac{\begin{array}{c}\vdots\,\pi\\ \mathbf{N}(y)\to A[\mathbf{suc}(y)] \quad A[\mathbf{0}] \quad \mathbf{N}(t)\end{array}}{A[t]}\,\mathrm{Sel}(\mathbf{N})$$

$$\frac{\forall y.A[y], \mathbf{N}(y)\to A[\mathbf{suc}(y)] \quad A[\mathbf{0}]}{\forall x.\mathbf{N}(x)\to A[x]}\,\mathrm{Ind}(\mathbf{N})$$

Restriction d'application des règles dans une dérivation :

- Dans la règle Sel(**N**), la dérivation π n'emploie pas la règle $\forall E$. y est une variable libre qui n'apparait ni dans les prémisses, ni dans la conclusion $A[t]$.

FIG. 6.2 – Règle d'introduction et d'élimination des entiers.

6.1.2 Les entiers

Dans l'esprit du livre de la théorie des types de Martin-Löf [ML84], nous définissons **ST**(**N**) en ajoutant au langage de **ST** un prédicat unaire **N** pour représenter la structure de données $\langle \mathbf{nat},\ \mathbf{0}:\mathbf{nat},\ \mathbf{suc}:\mathbf{nat}\to\mathbf{nat}\rangle$ des entiers bâtons, avec les règles indiquées par la figure 6.2. Les règles d'introduction précisent la construction des entiers.

Les règles d'élimination reflètent les propriétés de calcul que nous associons aux entiers. La règle d'induction traduit directement le principe usuel d'induction[10] qui correspond à la récursion primitive.

$$A[\mathbf{0}], (\forall y.\mathbf{N}(y), A[y]\to A[\mathbf{suc}(y)])\to(\forall x.\mathbf{N}(x)\to A[x])$$

Il faut garder en mémoire que le quantificateur universel filtre les instantiations, au travers la règle d'élimination $\forall E$. Dés lors, le paramètre d'induction sera nécessairement un terme canonique.

La règle de sélection exprime qu'un entier est soit **0** soit $\mathbf{suc}(t)$, *i.e.* correspond à l'axiome $x = \mathbf{0} \wedge x = \mathbf{suc}(\mathtt{pred}(x))$ où `pred` dénote la fonction prédecesseur $\mathtt{pred}(\mathbf{0}) = \mathbf{0}$ et $\mathtt{pred}(\mathbf{suc}(x)) = x$. La règle de sélection correspond à un branchement, dans une définition par cas.

Remarque 6.2. En retirant la restriction dans le domaine de la quantification, nous retrouvons le système **IT**(**N**) décrit par Leivant dans [Lei90, Lei00]. Dans ce cas, la règle de sélection n'est plus nécessaire car elle devient un cas particulier de la règle d'induction. Les fonctions prouvablement totales dans **IT**(**N**) sont exactement les fonctions prouvablement totales dans l'arithmétique de Peano.

[10] Comme dans les autres chapitres, nous écrivons $\tau_1,\ldots,\tau_n\to\tau$ pour $\tau_1\to(\tau_2\to\ldots\to\tau)$

$$\frac{}{\mathbf{W}(\boldsymbol{\epsilon})}\,\boldsymbol{\epsilon}I \qquad \frac{\mathbf{W}(t)}{\mathbf{W}(\mathbf{0}(t))}\,\mathbf{0}I \qquad \frac{\mathbf{W}(t)}{\mathbf{W}(\mathbf{1}(t))}\,\mathbf{1}I$$

$$\frac{\begin{array}{cccc}\vdots\,\pi_{\mathbf{0}} & \vdots\,\pi_{\mathbf{1}} & & \\ \mathbf{W}(y)\to A[\mathbf{0}(y)] & \mathbf{W}(y)\to A[\mathbf{1}(y)] & A[\boldsymbol{\epsilon}] & \mathbf{W}(t)\end{array}}{A[t]}\,\mathrm{Sel}(\mathbf{W})$$

$$\frac{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{0}(y)] \quad \forall y.A[y],\mathbf{W}(y)\to A[\mathbf{1}(y)] \quad A[\boldsymbol{\epsilon}]}{\forall x.\mathbf{W}(x)\to A[x]}\,\mathrm{Ind}(\mathbf{W})$$

Restriction d'application des règles dans une dérivation :

- Dans la règle Sel(**W**), les dérivations π_0 et π_1 n'emploient pas la règle $\forall E$. y est une variable libre qui n'apparait ni dans les prémisses, ni dans la conclusion $A[t]$.

FIG. 6.3 – Règle d'introduction et d'éliminations de $\mathbf{ST}(\mathbf{W})$.

6.1.3 Les mots

Nous venons de voir comment incorporer les entiers, à travers le système $\mathbf{ST}(\mathbf{N})$. De façon similaire, nous définissons $\mathbf{ST}(\mathbf{W})$ comme une extension de **ST** pour raisonner sur les mots de la structure $\langle$**word**, $\boldsymbol{\epsilon}$:**word**, **0**:**word**, **1**:**word**$\rangle$ en ajoutant les règles de la figure 6.3.

6.1.4 Sur les sortes

Etant donnée une sorte **s**, le système $\mathbf{ST}(\mathbf{s})$ est obtenu en incorporant

- Les règles d'introduction associées à chaque sorte, qui sont les règles de typage des constructeurs.
- Deux règles d'élimination obtenues automatiquement, comme dans la théorie des types de Martin-Löf : la règle d'induction correspond au schéma de récursion primitive, et la règle de sélection correspond à une définition par cas.

Enfin, $\mathbf{ST}(\mathcal{S})$ est le système qui comprend toutes les règles nécessaires à chaque $\mathbf{s} \in \mathcal{S}$. Nous ne détaillerons pas plus le cas général pour nous concentrer, essentiellement, sur la sorte **word**. En effet, la sorte **nat** est un cas particulier de cette dernière. Il est possible d'adapter chaque résultat à n'importe quelle sorte, ce que nous laissons au lecteur courageux.

6.1.5 Fonctions prouvablement totales

Les "preuves" de **ST** s'appellent des *dérivations*. Soit $\mathtt{f} : \mathbf{s}^n \to \mathbf{s}$ le programme $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \Gamma \rangle$. Définissons $\mathbf{ST}(\mathtt{f})$ le système $\mathbf{ST}(\mathbf{S})$ auquel nous ajoutons la règle de remplacement de la figure 6.4.

Une fonction ϕ est prouvablement totale dans une théorie H, si l'énoncé "Il existe e tel que pour tout x, il existe y, tel que le programme e sur l'entrée

$$\frac{A}{A[t \leftarrow v\theta]}\,\mathrm{R}$$

Restriction d'application des règles dans une dérivation
– A est un prédicat.
– θ est une substitution de $\mathcal{X} \to \mathcal{T}(\mathcal{C}, \mathcal{X})$, qui associe à chaque variable un terme canonique.
– $(v \to u) \in \mathcal{E}$ et $t = u\theta$.

FIG. 6.4 – Règle de remplacement.

x calcule y" est démontrable dans H. La formulation précise de cet énoncé dépend du système considéré.

Définition 6.3. Une fonction $\phi : [\![\mathbf{s}]\!]^n \to [\![\mathbf{s}]\!]$ est prouvablement totale dans **ST** s'il existe un programme $\mathtt{f}$ qui calcule ϕ, *i.e.* $\phi = [\![\mathtt{f}]\!]$ et il existe une dérivation dans $\mathbf{ST}(\mathtt{f})$, de

$$\mathrm{Tot}(\mathtt{f}) \equiv \forall \vec{x}.\mathbf{S}(x_1), \ldots, \mathbf{S}(x_n) \to \mathbf{S}(\mathtt{f}(x_1, \ldots, x_n))$$

Etant donnée une théorie H, quelles sont les fonctions prouvablement totales dans H ? Cette question, dont la motivation initiale plonge dans le problème des fondements des mathématiques, a été très étudiée. Suivant la formulation attribuée à Gödel-Kreisel, une fonction f est prouvablement totale dans H s'il existe une formule Π^0_2 telle que $\forall x.\exists y.T[x, y]$ qui est démontrable dans H et telle que pour tout n, $f(n) = \mu m.T[n, m]$. La comparaison entre ces deux formulations, et l'équivalence entre les ensembles de fonctions prouvablement totales, est faite dans l'article de Leivant [Lei00].

Pour l'étude des algorithmes, nous nous préoccupons de la question inverse. Etant donnée une classe de complexité C, existe-t-il une théorie H pour laquelle les fonctions prouvablement totales sont exactement les fonctions de C ? Il y a un lien étroit entre la complexité des fonctions de C et la complexité des formules d'induction, ou du schéma de compréhension. Nous nous restreignons au fragment suivant de **ST**.

Définition 6.4. Une formule A est une formule d'induction si $\forall x.\mathbf{S}(x) \to A[x]$ est la conclusion d'une induction. $\mathbf{ST}^0(\mathbf{S})$ est le système $\mathbf{ST}(\mathbf{S})$ dans lequel les formules d'induction sont des conjonctions de prédicats.

Exemple 6.5. Reprenons l'exemple classique de l'addition.

$$\mathtt{add}(\mathbf{0}, w) \to w$$
$$\mathtt{add}(\mathbf{suc}(x), w) \to \mathbf{suc}(\mathtt{add}(x, w))$$

La dérivation ci-dessous, que nous nommons $\pi_{\mathtt{add}}$, montre que l'addition est

une fonction prouvablement totale de $\mathbf{ST}(\mathtt{add})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{v{:}\mathbf{N}(\mathtt{add}(z,w))\}}{\mathbf{N}(\mathbf{suc}(\mathtt{add}(z,w)))}\,\mathbf{suc}I}{\mathbf{N}(\mathtt{add}(\mathbf{suc}(z),w))}\,\mathrm{R}}{\mathbf{N}(z),\mathbf{N}(\mathtt{add}(z,w))\to\mathbf{N}(\mathtt{add}(\mathbf{suc}(z),w))}\,{\to}I}{\forall z.\mathbf{N}(z),\mathbf{N}(\mathtt{add}(z,w))\to\mathbf{N}(\mathtt{add}(\mathbf{suc}(z),w))}\,\forall I \quad \dfrac{w{:}\mathbf{N}(w)}{\mathbf{N}(\mathtt{add}(\mathbf{0},w))}\,\mathrm{R}}{\forall x.\mathbf{N}(x)\to\mathbf{N}(\mathtt{add}(x,w))}\,\mathrm{Ind}(\mathbf{N})$$

Le terme w peut être remplacé par n'importe quel terme, canonique ou non. Poursuivons avec la multiplication dont les équations sont recopiées ci-dessous.

$$\begin{aligned}\mathtt{mul}(\mathbf{0},x) &\to \mathbf{0}\\ \mathtt{mul}(\mathbf{suc}(y),x) &\to \mathtt{add}(y,\mathtt{mul}(y,x))\end{aligned}$$

La multiplication est une fonction prouvablement totale de $\mathbf{ST}(\mathtt{mul})$ comme le montre la dérivation suivante.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{w{:}\mathbf{N}(\mathtt{mul}(z,x))\}}{\vdots\ \pi_{\mathtt{add}}[w\leftarrow\mathtt{mul}(z,x)]}}{\forall x.\mathbf{N}(x)\to\mathbf{N}(\mathtt{add}(x,\mathtt{mul}(z,x)))}}{\mathbf{N}(x)\to\mathbf{N}(\mathtt{add}(x,\mathtt{mul}(z,x)))}\,\forall E \quad \{x{:}\mathbf{N}(x)\}}{\mathbf{N}(\mathtt{add}(x,\mathtt{mul}(z,x)))}\,\mathrm{R}}{\mathbf{N}(z),\mathbf{N}(\mathtt{mul}(z,x))\to\mathbf{N}(\mathtt{mul}(\mathbf{suc}(z),x))}\,{\to}I}{\forall z.\mathbf{N}(z),\mathbf{N}(\mathtt{mul}(z,x))\to\mathbf{N}(\mathtt{mul}(\mathbf{suc}(z),x))}\,\forall I \quad \dfrac{\dfrac{}{\mathbf{N}(\mathbf{0})}\,\mathbf{0}I}{\mathbf{N}(\mathtt{mul}(\mathbf{0},x))}\,\mathrm{R}}{\dfrac{\forall y.\mathbf{N}(y)\to\mathbf{N}(\mathtt{mul}(y,x))}{\forall x.\forall y.\mathbf{N}(x),\mathbf{N}(y)\to\mathbf{N}(\mathtt{mul}(y,x))}\,(\forall E;{\to}I;\forall I)}$$

Remarque 6.6. Le raisonnement équationnel est traité par la réécriture, à la façon de la déduction modulo, voir l'habilitation de Dowek [Dow99]. A la différence de l'égalité extensionnelle, la réécriture évite des circonlocutions, qui semblent inutiles d'un point de vue algorithmique. Cependant, il serait possible de poursuivre la discussion en ajoutant

- le prédicat d'égalité $=$, avec ses axiomes.
- la clôture universelle de chacune des équations de $\mathcal{E}$. La clôture universelle de $\mathtt{f}(p_1,\ldots,p_n)\to t$ est $\forall\vec{x}.\mathtt{f}(p_1,\ldots,p_n)=t$, où les variables de $\mathtt{f}(p_1,\ldots,p_n)$ sont $\vec{x}=x_1,\ldots,x_m$.

La quantification limite, alors, la substitution aux termes canoniques. Le prix à payer est dans la fonction d'extraction du contenu calculatoire.

6.1.6 Espace linéaire

Théorème 6.7. *Les trois classes de fonctions suivantes sont identiques :*

1. *La classe des fonctions prouvablement totales dans* $\mathbf{ST}^0(\mathbf{N})$.
2. *La classe* $\mathcal{G}(2)$ *de Grzegorczyk des fonctions définies soit explicitement soit par récursion bornée par un polynôme.*

3. La classe des fonctions calculables en espace linéaire.

L'équivalence entre (2) et (3) est due à Ritchie [Rit63]. L'équivalence entre (1) et (3) est une conséquence du théorème 6.8.

Dans un article récent [ÇOW], Çağman, Ostrin et Wainer ont construit une arithmétique de Peano PA(;) dans lequel les arguments des prédicats sont divisés en arguments normaux et sûrs. L'arithmétique PA(;) est dans le prolongement de l'approche à la Bellantoni et Cook [BC92], que nous avons vu au chapitre 4. Ce système comporte, entre autre, des axiomes pour l'égalité et le prédecesseur. L'induction s'applique seulement aux termes normaux. La quantification ne porte que sur les arguments sûrs et est restreinte aux termes canoniques. Mais à la différence de **ST**, les axiomes de l'égalité permettent de relâcher la contrainte sur le domaine de quantification aux termes "héréditairement" canoniques. La classe de fonctions prouvablement totales dans le fragment de $PA(;)$ où l'induction est limitée aux formules $\Sigma_1(;)$ est exactement la classe $\mathcal{G}(2)$.

6.1.7 Temps polynomial

Théorème 6.8. *L'ensemble des fonctions prouvablement totales dans* $\mathbf{ST}^0(\mathbf{W})$ *est exactement la classe* PTIME *des fonctions calculables en temps polynomial.*

La démonstration est détaillée dans les sections suivantes.

6.1.8 Comparaison avec les autres approches

Buss [Bus86], et son école, ont développé des théories arithmétiques à quantification numériquement bornée (*i.e.* de la forme $\forall x < t$) et ont réussi à modéliser les principales classes de complexité. Non loin, citons la "bounded linear logic" de Girard, Scedrov et Scott dans [JYG92] qui comptabilise explicitement les ressources du calcul.

La première caractérisation purement syntaxique, qui est certainement le point de départ de la complexité implicite, est celle de Leivant dans [Lei91, Lei94a]. Leivant a démontré que les fonctions définies par des équations qui sont prouvablement totales dans l'arithmétique du second ordre avec un axiome de compréhension limité aux formules positives sans quantificateur , sont exactement les fonctions calculables en temps polynomial, *i.e.* PTIME. Leivant a ensuite transposé ce résultat au premier ordre dans l'article [Lei94b]. Pour cela, il a introduit une famille de prédicat $\mathbf{N}_0, \mathbf{N}_1, \ldots$ pour générer des entiers de valence $0, 1, \ldots$. D'un point de vue méthodologique, notre présentation est très similaire à celle de Leivant.

Girard a conçu la "light linear logic" comme une logique de temps polynomiale. Il s'agit d'une logique, à part entière, du second ordre, qui est affiliée à la logique linéaire. Les règles structurelles (affaiblissement, contraction) sont supprimées. Au lieu de cela, des modalités permettent de contrôler les ressources. Asperti[Asp98] et Roversi[Rov99] ont poli le travail originel de Girard.

Enfin, Bellantoni et Hofmann[BH00], ainsi que Schwichtenberg[Sch], ont emprunté l'idée des modalités de la logique linéaire, tout en se maintenant dans le cadre des systèmes arithmétiques, toujours pour caractériser PTIME.

6.2 Dérivation, normalisation et confluence

Dans une dérivation, certaines hypothèses sont rendues muettes et sont écrites entre accolades, *e.g.* $\{x:A\}$. Les autres hypothèses sont des *prémisses*. Nous utiliserons la notation $x_1:\mathbf{S}(t_1),\ldots,x_n:\mathbf{S}(t_n)\vdash A$ pour dire que A est la conclusion d'une dérivation π qui a pour prémisses $x_1:\mathbf{S}(t_1),\ldots,x_n:\mathbf{S}(t_n)$. Nous noterons $\pi[x\leftarrow s]$ la dérivation qui est obtenue en remplaçant la variable libre x par le terme s. Dans la suite, il faut comprendre, et cela est à la fois le point clef et la source des difficultés, que $\pi[x\leftarrow s]$ est toujours une dérivation de $\mathbf{ST}(\mathbf{S})$ si s est un terme canonique. Dans le cas où s n'est pas un terme canonique, suivant-le rôle de la prémisse $\mathbf{S}(t_i)$, il se peut que la dérivation ne soit plus une dérivation de $\mathbf{ST}(\mathbf{S})$. Le danger provient d'une dérivation π de $\mathbf{ST}(\mathbf{S})$ comme celle-ci :

$$\dfrac{\dfrac{\dfrac{\pi'}{\forall x.\mathbf{S}(x)\to A}\,\mathrm{Ind}(\mathbf{S})}{\mathbf{S}(t_i)\to A[x\leftarrow t_i]}\,\forall E \qquad \mathbf{S}(t_i)}{A[x\leftarrow t_i]}\to E$$

Ici, t_i est nécessairement un terme canonique, à cause de la règle $\forall E$. Si par mégarde, nous remplaçons x par un terme s non-canonique, la dérivation $\pi[x\leftarrow s]$ sortirait de $\mathbf{ST}(\mathbf{S})$.

Parmi les prémisses d'une règle d'élimination, appelons *majeure* celle qui contient le symbole à éliminer. Une *coupure* apparaît dans une dérivation quand une formule A est à la fois la conclusion d'une introduction ou une prémisse, et la majeure d'une élimination. L'autre occurrence de A est nommée la *mineure*. Une dérivation sans coupure est dite *normale*. Une coupure est un détour dans une preuve, qui correspond à l'application d'un lemme dans une démonstration. Gentzen [Gen35] a démontré que les coupures d'une dérivation pouvaient être supprimées dans le calcul des séquents et Prawitz [Pra65] en déduction naturelle. La liste des coupures et des conversions est précisée dans la figure 6.5 pour les connecteurs logiques, et dans les figures 6.6 et 6.7 pour les mots. Nous n'avons pas présenté les conversions pour les entiers, car c'est un cas particulier des mots.

	Coupures		*Conversions*
$\boldsymbol{\beta}$	$\dfrac{\dfrac{\begin{array}{c}\{x:A\}\\ \vdots\,\pi_0\\ B\end{array}}{A\to B}\to I \qquad \begin{array}{c}\vdots\,\pi_1\\ A\end{array}}{B}\to E$	$\triangleright$	$\begin{array}{c}\vdots\,\pi_1\\ A\\ \vdots\,\pi_0\\ B\end{array}$
$\wedge_{\boldsymbol{i}}$	$\dfrac{\dfrac{\begin{array}{cc}\vdots\,\pi_1 & \vdots\,\pi_2\\ A_1 & A_2\end{array}}{A_1\wedge A_2}\wedge I}{A_i}\wedge E$	$\triangleright$	$\begin{array}{c}\vdots\,\pi_i\\ A_i\end{array}$
$\forall$	$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi\\ A\end{array}}{\forall x.A}\forall I}{A[x\leftarrow p]}\forall E,\ p\in\mathcal{T}(\mathcal{C},\mathcal{X})$	$\triangleright$	$\begin{array}{c}\vdots\,\pi[x\leftarrow p]\\ A[x\leftarrow p]\end{array}$

FIG. 6.5 – Coupures et conversions de **ST**.

Sel

$$\dfrac{\mathbf{W}(x)\to A[\mathbf{0}(x)] \quad \mathbf{W}(x)\to A[\mathbf{1}(x)] \quad A[\boldsymbol{\epsilon}] \quad \mathbf{W}(\boldsymbol{\epsilon})}{A[\boldsymbol{\epsilon}]}\,\mathrm{Sel}(\mathbf{W}) \quad \triangleright \quad \dfrac{\mathbf{W}(x)\to A[\mathbf{0}(x)] \quad \mathbf{W}(x)\to A[\mathbf{1}(x)] \quad A[\boldsymbol{\epsilon}] \quad \dfrac{\mathbf{W}(t)}{\mathbf{W}(\mathbf{i}(t))}\,\mathbf{i}I}{A[\mathbf{i}(t)]}\,\mathrm{Sel}(\mathbf{W})$$

$$\triangleright \quad \dfrac{\mathbf{W}(t)\to A[\mathbf{i}(t)] \quad \dfrac{\pi_1[x\leftarrow t] \quad \pi_t}{\mathbf{W}(t)}\,{\to}E}{A[\mathbf{i}(t)]}$$

FIG. 6.6 – Coupures et Conversions de la règle de sélection de $\mathbf{ST}(\mathbf{W})$.

Rec

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_{\mathbf{0}}}{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{0}(y)]}\quad\dfrac{\vdots\,\pi_{\mathbf{1}}}{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{1}(y)]}\quad\dfrac{\vdots\,\pi_{\boldsymbol{\epsilon}}}{A[\boldsymbol{\epsilon}]}}{\dfrac{\forall x.\mathbf{W}(x)\to A[x]}{\mathbf{W}(\boldsymbol{\epsilon})\to A[\boldsymbol{\epsilon}]}\forall E}\,\mathrm{Ind}(\mathbf{W})\qquad\dfrac{\vdots\,\pi_{\boldsymbol{\epsilon}}}{\mathbf{W}(\boldsymbol{\epsilon})}}{A[\boldsymbol{\epsilon}]}\to E$$

$$\triangleright\quad\dfrac{\vdots\,\pi_{\boldsymbol{\epsilon}}}{A[\boldsymbol{\epsilon}]}$$

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_{\mathbf{0}}}{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{0}(y)]}\quad\dfrac{\vdots\,\pi_{\mathbf{1}}}{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{1}(y)]}\quad\dfrac{\vdots\,\pi_{\boldsymbol{\epsilon}}}{A[\boldsymbol{\epsilon}]}}{\dfrac{\forall x.\mathbf{W}(x)\to A[x]}{\mathbf{W}(\mathbf{i}(p))\to A[\mathbf{i}(p)]}\forall E}\,\mathrm{Ind}(\mathbf{W})\qquad\dfrac{\dfrac{\vdots\,\pi_p}{\mathbf{W}(p)}}{\mathbf{W}(\mathbf{i}(p))}\,\mathbf{i}I}{A[\mathbf{i}(p)]}\to E$$

$$\triangleright$$

$$\dfrac{\dfrac{\vdots\,\pi_{\mathbf{i}}}{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{i}(y)]}}{A[p],\mathbf{W}(p)\to A[\mathbf{i}(p)]}\forall E\qquad\dfrac{\dfrac{\dfrac{\vdots\,\pi_{\mathbf{0}}}{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{0}(y)]}\quad\dfrac{\vdots\,\pi_{\mathbf{1}}}{\forall y.A[y],\mathbf{W}(y)\to A[\mathbf{1}(y)]}\quad\dfrac{\vdots\,\pi_{\boldsymbol{\epsilon}}}{A[\boldsymbol{\epsilon}]}}{\dfrac{\forall x.\mathbf{W}(x)\to A[x]}{\mathbf{W}(p)\to A[p]}\forall E}\,\mathrm{Ind}(\mathbf{W})\qquad\dfrac{\vdots\,\pi_p}{\mathbf{W}(p)}}{A[p]}\to E\qquad\dfrac{\vdots\,\pi_p}{\mathbf{W}(p)}$$

$$\dfrac{\qquad\qquad\qquad\qquad}{A[\mathbf{i}(p)]}\to E$$

FIG. 6.7 – Coupures et conversions de la règle d'induction de $\mathbf{ST}(\mathbf{W})$

Proposition 6.9. *Soit π une dérivation de* $\mathbf{ST}(\mathbf{W})$ *telle que $\pi \triangleright \pi'$. Alors, π' est une dérivation de* $\mathbf{ST}(\mathbf{W})$.

Démonstration. Dans le cas d'une conversion d'une induction $\mathrm{Ind}(\mathbf{W})$, nous voyons que $\mathbf{i}(p)$ est un terme canonique car π est une dérivation de $\mathbf{ST}(\mathbf{W})$. De ce fait π' est une dérivation de $\mathbf{ST}(\mathbf{W})$ d'après la remarque qui précède. Dans le cas d'une conversion d'une sélection $\mathrm{Sel}(\mathbf{W})$, t n'est pas nécessairement canonique. Mais, $\pi_{\mathbf{i}}$ ne contient pas la règle $(\forall E)$, donc $\pi_{\mathbf{i}}[x \leftarrow t]$ est dans $\mathbf{ST}(\mathbf{W})$. Enfin, il est facile de constater que pour chaque autre conversion, la dérivation π' est dans $\mathbf{ST}(\mathbf{W})$. □

Théorème 6.10. $\mathbf{ST}(\mathbf{W})$ *est fortement normalisable,* i.e. $\triangleright$ *est nœthérien.*

Démonstration. D'après la proposition précédente, la conversion $\triangleright$ ne fait pas sortir de $\mathbf{ST}(\mathbf{W})$. D'autre part, chaque conversion de $\mathbf{ST}(\mathbf{W})$ est une conversion de $\mathbf{IT}(\mathbf{W})$. Puisque $\mathbf{IT}(\mathbf{W})$ est fortement normalisable, d'après [Lei00], il s'ensuit que $\mathbf{ST}(\mathbf{W})$ l'est aussi. □

Théorème 6.11. $\mathbf{ST}(\mathbf{W})$ *est confluent,* i.e. $\triangleright$ *est confluente.*

6.3 Synthèse de programmes et complexité

6.3.1 Méthodologie

Dans la sémantique de Heyting, l'interprétation d'une formule A est l'ensemble de ses dérivations. La correspondance κ de Curry-Howard, ou la réalisabilité de Kleene [Kle52], interprète une dérivation comme un programme qui réalise A. Le programme équationnel `f` de `E` est conçu comme une spécification algébrique qui dénote la fonction $[\![\mathtt{f}]\!]$. La compilation de `f` comporte deux phases. La première phase consiste à construire une dérivation π de la preuve de totalité $\mathrm{Tot}(\mathtt{f})$ dans $\mathbf{ST}(\mathtt{f})$. La seconde phase extrait de la dérivation π un programme $\kappa(\pi)$ qui calcule $[\![\mathtt{f}]\!]$.

Ainsi, l'existence d'une dérivation π implique que le programme $\kappa(\pi)$ implémente correctement la spécification `f`. Des environnements de démonstration/synthèse de programmes tels que Alf [Alf], Coq [Coq], Nuprl [Ca86] permettent de compiler un *programme certifié* à partir d'une spécification. Comme le souligne Constable [Con95], l'efficacité du programme ainsi compilé $\kappa(\pi)$ n'est pas assurée. Ce travail apporte une analyse de la complexité du programme $\kappa(\pi)$ à partir de sa dérivation. Nous nous situons à mi-chemin entre la programmation par la théorie des types de Martin-löf, telle qu'elle est exposée dans le livre de Nordström, Peterson et Smith [NPS90], et la programmation dans le cadre de AF_2 de Leivant [Lei83] et étudiée par Krivine et Parigot [KP87, Kri90, Par92].

6.3.2 Langage fonctionnel d'ordre supérieur

Notre point de départ est le λ-calcul. Nous renvoyons le lecteur aux ouvrages bien connus comme le livre de Barendregt [Bar80], ou de Hindley et Seldin [HS86] pour un cours complet sur le λ-calcul. La syntaxe des termes est

donnée par la grammaire suivante :

$$\begin{array}{lllll} \textit{(Variables)} & \mathcal{X} \ni x & ::= & x \mid y \mid z \mid \ldots \\ \textit{(Termes)} & \mathcal{T}(\mathcal{C},\mathcal{F},\mathcal{X}) \ni t & ::= & p \mid \lambda x.t \mid t(s) \mid \mathbf{pr}_1 \mid \mathbf{pr}_2 \mid \langle \_,\_ \rangle \end{array}$$

Les réductions de termes sont

$$\begin{array}{lll} \boldsymbol{\beta} & (\lambda x.t)s & \rightarrow\ t[x \leftarrow s] \\ \boldsymbol{\wedge_i} & \mathbf{pr}_i(\langle t_1, t_2\rangle) & \rightarrow\ t_i \end{array}$$

L'interprétation $[\![t]\!]^{\boldsymbol{\lambda}}$ d'un terme t est la forme normale de t si elle existe, *i.e.* $[\![t]\!]^{\boldsymbol{\lambda}} = \mathbf{a}$ ssi $t \xrightarrow{!} \mathbf{a}$, en suivant les notations du chapitre 1.

6.3.3 Le cas des entiers

Le langage $\boldsymbol{\lambda}(\mathbf{nat})$ est un λ-calcul non-typé dont le caractère opérationnel est identique au système T de Gödel.

La grammaire des termes de $\boldsymbol{\lambda}(\mathbf{nat})$ est augmentée par

$$\begin{array}{lllll} \textit{(Constructeurs)} & \mathcal{C} \ni \mathbf{c} & ::= & \mathbf{0} \mid \mathbf{suc} \\ \textit{(Constantes)} & \mathbf{r} & ::= & \mathbf{natrec} \mid \mathbf{natcase} \end{array}$$

Chaque terme de $\boldsymbol{\lambda}(\mathbf{nat})$ est un programme sur $[\![\mathbf{nat}]\!]$. La fonction sémantique $[\![\ ]\!]^{\boldsymbol{\lambda}}$ est prolongée par les règles de réductions de termes suivantes :

$$\begin{array}{lrl} \mathbf{Sel} & \mathbf{natcase}(t_{\mathbf{suc}})(t_{\mathbf{0}})(\mathbf{0}) & \rightarrow t_{\mathbf{0}} \\ & \mathbf{natcase}(t_{\mathbf{suc}})(t_{\mathbf{0}})(\mathbf{suc}(u)) & \rightarrow t_{\mathbf{suc}}(u) \\ \mathbf{Rec} & \mathbf{natrec}(t_{\mathbf{suc}})(t_{\mathbf{0}})(\mathbf{0}) & \rightarrow t_{\mathbf{0}} \\ & \mathbf{natrec}(t_{\mathbf{suc}})(t_{\mathbf{0}})(\mathbf{suc}(u)) & \rightarrow t_{\mathbf{suc}}(\mathbf{natrec}(t_{\mathbf{suc}})(t_{\mathbf{0}})(u))(u) \end{array}$$

6.3.4 Le cas des mots

De même, la grammaire des termes $\boldsymbol{\lambda}(\mathbf{word})$ contient :

$$\begin{array}{lllll} \textit{(Constructeurs)} & \mathcal{C} \ni \mathbf{c} & ::= & \boldsymbol{\epsilon} \mid \mathbf{0} \mid \mathbf{1} \\ \textit{(Constantes)} & \mathbf{r} & ::= & \mathbf{binrec} \mid \mathbf{bincase} \end{array}$$

Les règles de réductions sont

$$\begin{array}{lrl} \mathbf{Sel} & \mathbf{bincase}(t_{\mathbf{0}})(t_{\mathbf{1}})(t_{\boldsymbol{\epsilon}})(\boldsymbol{\epsilon}) & \rightarrow t_{\boldsymbol{\epsilon}} \\ & \mathbf{bincase}(t_{\mathbf{0}})(t_{\mathbf{1}})(t_{\boldsymbol{\epsilon}})(\mathbf{i}(u)) & \rightarrow t_{\mathbf{i}}(u) \\ \mathbf{Rec} & \mathbf{binrec}(t_{\mathbf{0}})(t_{\mathbf{1}})(t_{\boldsymbol{\epsilon}})(\boldsymbol{\epsilon}) & \rightarrow t_{\boldsymbol{\epsilon}} \\ & \mathbf{binrec}(t_{\mathbf{0}})(t_{\mathbf{1}})(t_{\boldsymbol{\epsilon}})(\mathbf{i}(u)) & \rightarrow t_{\mathbf{i}}(\mathbf{binrec}(t_{\mathbf{i}})(t_{\boldsymbol{\epsilon}})(u))(u) \end{array}$$

6.3.5 Le cas général

Le système $\mathbf{ST}$ permet de raisonner sur les programmes équationnels de $\mathtt{E}$ quelque soient leurs domaines de calcul. Soit $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \Gamma \rangle$ un programme typé et $\mathcal{S}$ l'ensemble des sortes des constructeurs $\mathcal{C}$ suivant Γ (*i.e.* $\mathcal{S} = \mathrm{dom}(\Gamma)$). Le programme source $\mathtt{f}$ sera compilé par un programme du langage fonctionnel d'ordre supérieur $\boldsymbol{\lambda}(\mathcal{S})$, à partir d'une dérivation de $\mathrm{Tot}(\mathtt{f})$ de $\mathbf{ST}(\mathcal{S})$.

Le langage de programmation $\boldsymbol{\lambda}(\mathcal{S})$ en ajoutant au λ-calcul

– les constructeurs de $\mathcal{C}$

– deux constantes **srec** et **scase** pour chaque sorte $\mathbf{s} \in \mathcal{S}$.

6.3.6 Extraction de programme

Nous allons définir l'extracteur κ qui synthétise dans $\boldsymbol{\lambda}(\mathbf{s})$ le contenu algorithmique de chaque dérivation de $\mathbf{ST}(\mathbf{S})$. La définition de l'extracteur κ est donnée par la figure 6.8 pour la partie logique, par la figure 6.9 pour les entiers et par la figure 6.10 pour les mots.

Exemple 6.12. Le programme $\kappa(\pi_{\mathtt{add}})$ extrait de la dérivation $\pi_{\mathtt{add}}$ est

$$\kappa(\pi_{\mathtt{add}})[w] \equiv \lambda x.\mathbf{natrec}(\lambda z\lambda v.\mathbf{suc}(v))(w)(x)$$

De ce fait, le programme qui réalise la spécification de `add`, est

$$\lambda w\lambda x.\mathbf{natrec}(\lambda z\lambda v.\mathbf{suc}(v))(x)(w)$$

Le programme de la multiplication `mul` est

$$\kappa(\pi_{\mathtt{mul}}) \equiv \lambda x\lambda y.\mathbf{natrec}(\lambda z.\lambda w.\kappa(\pi_{\mathtt{add}})[w](x))(\mathbf{0})(y)$$

Remarque 6.13. Le terme $\kappa(\pi)$ de $\boldsymbol{\lambda}(\mathbf{nat})$ est un terme du système T car il est typable. En effet, nous associons à chaque formule A du premier ordre un type $\underline{A}$ comme suit : $\underline{\mathbf{N}} = \mathbf{nat}$, $\underline{A \to B} = \underline{A} \to \underline{B}$, $\underline{A \wedge B} = \underline{A} \wedge \underline{B}$, $\underline{\forall x.A} = \underline{A}$. Le terme $\kappa(\pi)$ a pour type $\underline{A}$ où A est la conclusion de la dérivation π.

Lemme 6.14. *Soit π une dérivation normale de $\mathbf{S}(t)$ où $t \in \mathcal{T}(\mathcal{C}, \mathcal{F})$ est un terme clos. Nous avons $[\![\kappa(\pi)]\!]^{\boldsymbol{\lambda}} = \kappa(\pi) = [\![t]\!]$.*

Démonstration. Puisque π est une dérivation normale de $\mathbf{S}(t)$, les seules règles employées sont celles des introductions de constructeurs, et la règle de substitution. La démonstration est immédiate par induction sur la taille de π. □

Le lemme suivant établit que l'extracteur est un homomorphisme de l'ensemble des dérivations de $\mathbf{ST}(\mathbf{S})$ muni des conversions ($\triangleright$) de coupures dans l'ensemble des programmes de $\boldsymbol{\lambda}(\mathbf{s})$ muni des réductions ($\to$) de termes.

Lemme 6.15. *Supposons que $t = \kappa(\pi) \to t'$. Il existe une dérivation π' telle que $t' = \kappa(\pi')$ et $\pi \triangleright \pi'$. Réciproquement, si $\pi \triangleright \pi'$, alors $\kappa(\pi) \to \kappa(\pi')$.*

Démonstration. Par induction sur la taille de π. □

Théorème 6.16. *Soit π une dérivation de $\mathrm{Tot}(\boldsymbol{f})$. Le programme $\kappa(\pi)$ calcule la fonction $[\![f]\!]$ dans le sens suivant : Pour tout $\mathbf{a}_1, \ldots, \mathbf{a}_n \in [\![\mathbf{s}]\!]$, la forme normale de $\kappa(\pi)(\mathbf{a}_1)\ldots(\mathbf{a}_n)$ est $[\![f]\!](\mathbf{a}_1, \ldots, \mathbf{a}_n)$, autrement dit*

$$[\![\kappa(\pi)(\mathbf{a}_1)\ldots(\mathbf{a}_n)]\!]^{\boldsymbol{\lambda}} = [\![f]\!](\mathbf{a}_1, \ldots, \mathbf{a}_n)$$

Démonstration. Pour simplifier la démonstration, nous supposons que `f` est d'arité 1. Considérons la dérivation π' suivante :

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi}{\forall x.\mathbf{S}(x) \to \mathbf{S}(\mathtt{f}(x))}\,\forall E}{\mathbf{S}(\mathbf{a}) \to \mathbf{S}(\mathtt{f}(\mathbf{a}))} \qquad \dfrac{\vdots\,\pi_{\mathbf{a}}}{\mathbf{S}(\mathbf{a})}}{\mathbf{S}(\mathtt{f}(\mathbf{a}))}\,\to E$$

dans laquelle $\pi_{\mathbf{a}}$ est une dérivation normale.

Dérivation π	$\overset{\kappa}{\Rightarrow}$	Terme $\kappa(\pi)$
$x : A$		x
$\dfrac{\begin{array}{c}\{x:A\}\\ \vdots\,\pi\\ B\end{array}}{A \to B} \to I$		$\lambda x.\kappa(\pi)$
$\dfrac{\begin{array}{cc}\vdots\,\pi_1 & \vdots\,\pi_2\\ A \to B & A\end{array}}{B} \to E$		$\kappa(\pi_1)(\kappa(\pi_2))$
$\dfrac{\begin{array}{cc}\vdots\,\pi_1 & \vdots\,\pi_2\\ A_1 & A_2\end{array}}{A_1 \wedge A_2} \wedge I$		$\langle \kappa(\pi_1)\kappa(\pi_2)\rangle$
$\dfrac{\begin{array}{c}\vdots\,\pi\\ A_1 \wedge A_2\end{array}}{A_i} \wedge E$		$\mathbf{pr}_i(\kappa(\pi))$
$\dfrac{\begin{array}{c}\vdots\,\pi\\ A\end{array}}{\forall x.A} \forall I$		$\kappa(\pi)$
$\dfrac{\begin{array}{c}\vdots\,\pi\\ \forall x.A\end{array}}{A[t/x]} \forall E$		$\kappa(\pi)$

FIG. 6.8 – L'extracteur κ de **ST**.

$$\dfrac{\begin{array}{ccc}\vdots\,\pi_{\mathbf{suc}} & \vdots\,\pi_{\mathbf{0}} & \vdots\,\pi_t\\ \mathbf{N}(y)\to A[\mathbf{suc}(y)] & A[\mathbf{0}] & \mathbf{N}(t)\end{array}}{A[t]}\ \mathrm{Sel}(\mathbf{N})$$

$$\overset{\kappa}{\Rightarrow}\qquad \mathbf{natcase}(\kappa(\pi_{\mathbf{suc}}))(\kappa(\pi_{\mathbf{0}}))(\kappa(\pi_t))$$

$$\dfrac{\begin{array}{cc}\vdots\,\pi_{\mathbf{suc}} & \vdots\,\pi_{\mathbf{0}}\\ \forall y.A[y],\mathbf{N}(y)\to A[\mathbf{suc}(y)] & A[\mathbf{0}]\end{array}}{\forall x.\mathbf{N}(x)\to A[x]}\ \mathrm{Ind}(\mathbf{N})$$

$$\overset{\kappa}{\Rightarrow}\qquad \lambda x.\mathbf{natrec}(\kappa(\pi_{\mathbf{0}}))(\kappa(\pi_{\mathbf{suc}}))(x)$$

FIG. 6.9 – L'extracteur κ de $\mathbf{ST}(\mathbf{N})$.

$$\dfrac{\begin{array}{cccc}\vdots\,\pi_{\mathbf{0}} & \vdots\,\pi_{\mathbf{1}} & \vdots\,\pi_{\boldsymbol{\epsilon}} & \vdots\,\pi_t\\ \mathbf{W}(y)\to A[\mathbf{0}(y)] & \mathbf{W}(y)\to A[\mathbf{1}(y)] & A[\boldsymbol{\epsilon}] & \mathbf{W}(t)\end{array}}{A[t]}\ \mathrm{Sel}(\mathbf{W})$$

$$\overset{\kappa}{\Rightarrow}\qquad \mathbf{bincase}(\kappa(\pi_{\mathbf{0}}),(\kappa(\pi_{\mathbf{1}},\kappa(\pi_{\boldsymbol{\epsilon}}),\kappa(\pi_t))$$

$$\dfrac{\begin{array}{ccc}\vdots\,\pi_{\mathbf{0}} & \vdots\,\pi_{\mathbf{1}} & \vdots\,\pi_{\boldsymbol{\epsilon}}\\ \forall y.A[y],\mathbf{W}(y)\to A[\mathbf{0}(y)] & \forall y.A[y],\mathbf{W}(y)\to A[\mathbf{1}(y)] & A[\boldsymbol{\epsilon}]\end{array}}{\forall x.\mathbf{W}(x)\to A[x]}\ \mathrm{Ind}(\mathbf{W})$$

$$\overset{\kappa}{\Rightarrow}\qquad \lambda x.\mathbf{binrec}(\kappa(\pi_{\mathbf{0}}))(\kappa(\pi_{\mathbf{1}}))(\kappa(\pi_{\boldsymbol{\epsilon}}))(x)$$

FIG. 6.10 – L'extracteur κ de $\mathbf{ST}(\mathbf{W})$.

Le lemme 6.14 implique $\kappa(\pi_{\mathbf{a}}) = [\![\mathbf{a}]\!] = \mathbf{a}$, car $\mathbf{a} \in [\![\mathbf{s}]\!]$. Donc, par définition de κ, $\kappa(\pi') = \kappa(\pi)\kappa(\mathbf{a}) = \kappa(\pi)\mathbf{a}$. D'après le théorème 6.10, nous savons que π' possède une forme normale δ. D'autre part, $\kappa(\delta)$ est la forme normale de $\kappa(\pi') = \kappa(\pi)\mathbf{a}$, d'après le lemme 6.15. En ré-appliquant le lemme 6.14, nous obtenons $[\![\kappa(\delta)]\!]^{\boldsymbol{\lambda}} = [\![\kappa(\pi)\mathbf{a}]\!]^{\boldsymbol{\lambda}} = [\![\mathtt{f}(\mathbf{a})]\!] = [\![\mathtt{f}]\!](\mathbf{a})$. □

6.3.7 Complexité des programmes extraits

Théorème 6.17. *Soit $\phi : [\![\mathbf{word}]\!]^n \to [\![\mathbf{word}]\!]$ une fonction prouvablement totale dans $\mathbf{ST}^0(\mathbf{W})$. La fonction ϕ est calculable en temps polynomial dans la somme de la taille des entrées.*

Démonstration. D'après les hypothèses du théorème, il existe un programme $\mathtt{f}$ qui calcule ϕ, *i.e.* $\phi = [\![\mathtt{f}]\!]$ et tel que $\mathrm{Tot}(\mathtt{f})$ est dérivable dans $\mathbf{ST}^0(\mathtt{f})$. Pour ne pas compliquer inutilement la démonstration, nous supposerons que $\mathrm{Tot}(\mathtt{f})$ a une dérivation qui n'utilise pas la règle $\wedge E$.

En préalable, disons qu'un prédicat de sorte $\mathbf{W}(p)$ est un *prédicat de récurrence*, dans une dérivation π si

(a) s'il existe une occurence de $\mathbf{W}(p)$ qui est dans la partie négative de la conclusion d'une induction.

(b) s'il existe une occurence $\mathbf{W}(p)$ qui est la mineure d'une coupure sur une induction.

Un exemple typique est donné ci-dessous, dans lequel $\mathbf{W}(x)$ (cas (a)) et $\mathbf{W}(t)$, conclusion de π_t, (cas (b)) sont des prédicats de récurrence.

$$\dfrac{\dfrac{\dfrac{}{\forall x.\mathbf{W}(x) \to A[x]}\,\mathrm{Ind}(\mathbf{W})}{\mathbf{W}(t) \to A[t]}\,\forall E \qquad \begin{matrix}\vdots\,\pi_t\\ \mathbf{W}(t)\end{matrix}}{A[t]}\,{\to}E$$

Il faut noter que si $\mathbf{W}(t)$ est un prédicat de récurrence, alors t est nécessairement canonique.

La démonstration du théorème est, alors, une conséquence directe du fait qui suit.

Fait 6.18. Soit π une dérivation normale, sans la règle $\wedge E$, de

$$y_1 : \mathbf{W}(s_1), \ldots, y_m : \mathbf{W}(s_m) \vdash \forall \vec{x}.\mathbf{W}(s_{m+1}), \ldots, \mathbf{W}(s_{m+n}) \to \mathbf{W}(t)$$

où t n'est pas un terme canonique.

Pour tout $\mathbf{a}_1, \ldots, \mathbf{a}_{n+m} \in [\![\mathbf{word}]\!]$, le calcul de

$$\kappa(\pi)(\mathbf{a}_{m+1}) \ldots (\mathbf{a}_{m+n})[s_1 \leftarrow \mathbf{a}_1, \ldots, s_m \leftarrow \mathbf{a}_m] \qquad (*)$$

est effectué en temps borné par $p_\pi(M)$ et constant en N où

$$M = \max(|\mathbf{a}_i| : \mathbf{W}(s_i) \text{ est un prédicat de récurrence})$$
$$N = \max(|\mathbf{a}_i| : \mathbf{W}(s_i) \text{ n'est pas un prédicat de récurrence})$$

La démonstration est faite par récurrence sur la taille de la dérivation normale π. Nous examinons la dernière règle de π. Nous ne détaillerons que les trois cas qui posent des difficultés.

La dernière règle de π est une induction Ind($\mathbf{W}$), (nous posons $n = 1$ et $s_{m+1} = x$ dans l'énoncé (*))

$$\frac{\begin{array}{ccc}\vdots\,\pi_{\mathbf{0}} & \vdots\,\pi_{\mathbf{1}} & \vdots\,\pi_{\boldsymbol{\epsilon}}\\ \forall y.\mathbf{W}(t[y]), \mathbf{W}(y) \to \mathbf{W}(t[\mathbf{0}(y)]) & \forall y.\mathbf{W}(t[y]), \mathbf{W}(y) \to \mathbf{W}(t[\mathbf{1}(y)]) & \mathbf{W}(t[\boldsymbol{\epsilon}])\end{array}}{\forall x.\mathbf{W}(x) \to \mathbf{W}(t[x])}$$

Par définition, le terme $\kappa(\pi)$ est $\lambda x.\mathbf{binrec}(\kappa(\pi_{\mathbf{0}}))(\kappa(\pi_{\mathbf{1}}))(\kappa(\pi_{\boldsymbol{\epsilon}}))(x)$. Le calcul de $\kappa(\pi)\mathbf{a}_{m+1}[s_1 \leftarrow \mathbf{a}_1, \ldots, s_m \leftarrow \mathbf{a}_m]$ s'effectue par une itération en évaluant successivement

$$\begin{aligned} u_0 &= \kappa(\pi_0)[s_1 \leftarrow \mathbf{a}_1, \ldots, s_m \leftarrow \mathbf{a}_m] \\ u_{n+1} &= \kappa(\pi_{\mathbf{a}[n+1]})(\mathbf{a}[1 \ldots n])(u_n)[s_1 \leftarrow \mathbf{a}_1, \ldots, s_m \leftarrow \mathbf{a}_m] \end{aligned}$$

où le mot $\mathbf{a}_{m+1}$ s'écrit $\mathbf{a}[p](\ldots(\mathbf{a}[1](\boldsymbol{\epsilon})\ldots)$ et $\mathbf{a}[1 \ldots n]$ est le sous-mot $\mathbf{a}[1 \ldots n] = \mathbf{a}[n](\ldots(\mathbf{a}[1](\boldsymbol{\epsilon})\ldots)$.

Le résultat de l'évaluation de $\kappa(\pi)\mathbf{a}_{m+1}[s_1 \leftarrow \mathbf{a}_1, \ldots, s_m \leftarrow \mathbf{a}_m]$ est fourni par u_p. Or, $p \leq M$ car $\mathbf{W}(x)$ est un prédicat de récurrence. Les termes $\kappa(\pi_{\boldsymbol{\epsilon}})$ et $\kappa(\pi_{\mathbf{i}})$ vérifient les hypothèses du fait, ainsi que l'hypothèse d'induction. Par hypothèse du lemme, $t[y]$ n'est pas un terme canonique. Donc, $\mathbf{W}(t[y])$ n'est pas un paramètre de récurrence. Il s'ensuit que le temps de calcul est borné par $M \cdot \max(P_{\pi_{\mathbf{0}}}(M), P_{\pi_{\mathbf{1}}}(M)) + P_{\pi_{\boldsymbol{\epsilon}}}(M)$, et conatant en N.

La dernière règle est une sélection,

$$\frac{\begin{array}{cccc}\vdots\,\pi_{\mathbf{0}} & \vdots\,\pi_{\mathbf{1}} & \vdots\,\pi_{\boldsymbol{\epsilon}} & \vdots\,\pi_s\\ \mathbf{W}(y) \to \mathbf{W}(t[\mathbf{0}(y)]) & \mathbf{W}(y) \to \mathbf{W}(t[\mathbf{1}(y)]) & \mathbf{W}(t[\boldsymbol{\epsilon}]) & \mathbf{W}(s)\end{array}}{\mathbf{W}(t[s])}\,\mathrm{Sel}(\mathbf{W})$$

Le terme $\kappa(\pi)$ est $\mathbf{bincase}(\kappa(\pi_{\mathbf{0}}))(\kappa(\pi_{\mathbf{1}}))(\kappa(\pi_{\boldsymbol{\epsilon}}))(\kappa(\pi_t))$. Son évaluation consiste à calculer $\kappa(\pi_t)[s_1 \leftarrow \mathbf{a}_1, \ldots, s_m \leftarrow \mathbf{a}_m]$. Suivant le résultat obtenu, il faut brancher vers $\kappa(\pi_{\boldsymbol{\epsilon}})$ ou $\kappa(\pi_{\mathbf{i}})$. Le point important est que $\mathbf{W}(y)$ n'est pas un paramètre de récurrence car la dérivation $\pi_{\mathbf{i}}$ ne contient pas la règle $\forall E$. Nous concluons que le temps de calcul est borné par $\max(P_{\pi_{\mathbf{0}}}(M), P_{\pi_{\mathbf{1}}}(M), P_{\pi_{\boldsymbol{\epsilon}}}(M)) + P_{\pi_s}(M) + O(1)$, et constant en N.

La dernière règle est $\to E$. Du fait que la dérivation π est normale, l'élimination de l'implication résulte d'une induction.

$$\dfrac{\dfrac{\dfrac{}{\forall x.\mathbf{W}(x) \to \mathbf{W}(t[x])}\,\mathrm{Ind}(\mathbf{W})}{\mathbf{W}(p) \to \mathbf{W}(t[p])}\,\forall E \qquad \mathbf{W}(p)}{\mathbf{W}(t[p])} \to E$$

Puisque la dérivation π est normale, $\mathbf{W}(p)$ est une prémisse. Donc, ce cas revient à celui de l'induction, que nous avons déjà traité. □

6.4 Expressivité des fonctions prouvablement totales

Théorème 6.19. *Chaque fonction ϕ calculable en temps polynomial est prouvablement total dans* $\mathbf{ST}^0(\mathbf{W})$.

Démonstration. La fonction ϕ est calculée par une machine de Turing M en temps $c \cdot n^c$ où n est la taille des entrées et c une constante. L'alphabet du ruban de M est $\{0,1\}$ et l'ensemble Q des états est un sous-ensemble fini de $\{0,1\}^*$. A chaque étape, M est complètement décrite par un triplet (q,u,v) dans lequel q est l'état de la machine, u est la partie à gauche de la tête, et v est la partie à droite de la tête. La tête est positionnée sur la première lettre de v. La fonction de transition de M est $\delta : Q \times \{0,1\} \mapsto Q \times \{0,1\} \times \{l,r\}$. La description initiale de M est (q_0, ϵ, v). A la fin du calcul, le résultat est à droite de la tête.

La démonstration consiste à définir un programme de $\mathtt{E}$ sur le domaine de calcul $\langle \mathbf{word},\ \boldsymbol{\epsilon} : \mathbf{word},\ \mathbf{0} : \mathbf{word},\ \mathbf{1} : \mathbf{word} \rangle$ qui simule M, puis de montrer que ce programme définit une fonction prouvablement totale dans $\mathbf{ST}^0(\mathbf{W})$.

Tout d'abord, pour passer d'une description à l'étape t à la description suivante, à l'étape $t+1$, nous employons trois programmes $\mathtt{D}_0, \mathtt{D}_1$ et $\mathtt{D}_2$. Ces programmes indiquent, respectivement, le nouvel état, le mot à gauche, le mot à droite, à partir de la fonction de transition δ.

$$\begin{aligned}
\mathtt{D}_0(q,u,\mathbf{i}(v)) &\to q' && \text{si } \delta(q,i) = (q',k,_)\\
\mathtt{D}_1(q,\mathbf{j}(u),\mathbf{i}(v)) &\to u && \text{si } \delta(q,i) = (q',k,l)\\
\mathtt{D}_1(q,u,\mathbf{i}(v)) &\to \mathbf{k}(u) && \text{si } \delta(q,i) = (q',k,r)\\
\mathtt{D}_2(q,\mathbf{j}(u),\mathbf{i}(v)) &\to \mathbf{j}(\mathbf{k}(v)) && \text{si } \delta(q,i) = (q',k,l)\\
\mathtt{D}_2(q,u,\mathbf{i}(v)) &\to v && \text{si } \delta(q,i) = (q',k,r)
\end{aligned}$$

Dans les autres cas, le résultat des $\mathtt{D}_i$ est $\boldsymbol{\epsilon}$. Ainsi,

$$(\mathtt{D}_0(c_0,c_1,c_2), \mathtt{D}_1(c_0,c_1,c_2), \mathtt{D}_2(c_0,c_1,c_2))$$

est la description qui suit la description (c_0,c_1,c_2).

Puisque chaque $\mathtt{D}_i$ est défini par cas, nous pouvons construire une dérivation $\pi_{\mathtt{D}_i}$ de $\mathbf{W}(q), \mathbf{W}(u), \mathbf{W}(v) \vdash \mathbf{W}(\mathtt{D}_i(q,u,v))$ qui est obtenue en imbriquant les règles de Sel($\mathbf{W}$) pour déterminer l'état et les lettres voisines de la tête, sans utiliser la règle d'induction. Par conséquent, nous pourrons remplacer les variables q, u et v par n'importe quel terme, y compris un terme non-canonique.

Maintenant, nous définissons, par récursion mutuelle sur i et m, la suite $(\mathtt{L}^i_m(t,x,c_0,c_1,c_2))_{i=0,1,2}$:

$$\begin{aligned}
\mathtt{L}^i_0(t,x,c_0,c_1,c_2) &\to \mathtt{D}_i(c_0,c_1,c_2)\\
\mathtt{L}^i_{m+1}(\epsilon,x,c_0,c_1,c_2) &\to c_i\\
\mathtt{L}^i_{m+1}(\mathbf{j}(t),x,c_0,c_1,c_2) &\to \mathtt{L}^i_m(x,x,\Delta^0,\Delta^1,\Delta^2) && \mathbf{j} = \mathbf{0},\mathbf{1}
\end{aligned}$$

où $\Delta^i = \mathtt{L}^i_{m+1}(t,x,c_0,c_1,c_2)$, for $i = 0,1,2$.

La description de M après $|t| \cdot |x|^m$ étapes, à partir de la description (c_0,c_1,c_2), est $(\mathtt{L}^0_{m+1}(t,x,c_0,c_1,c_2), \mathtt{L}^1_{m+1}(t,x,c_0,c_1,c_2), \mathtt{L}^2_{m+1}(t,x,c_0,c_1,c_2))$. Le résultat de M, à partir de la description initiale (q_0,ϵ,v) après $c.|v|^c$ étapes, est $(\mathtt{L}^0_{c+1}(c,v,q_0,\boldsymbol{\epsilon},v), \mathtt{L}^1_{c+1}(c,v,q_0,\boldsymbol{\epsilon},v), \mathtt{L}^2_{c+1}(c,v,q_0,\boldsymbol{\epsilon},v))$. Nous en déduisons que la fonction ϕ, calculée par la machine M, est programmée par $\mathtt{f}$

$$\mathtt{f}(v) \to \mathtt{L}^2_{c+1}(c,v,q_0,\boldsymbol{\epsilon},v)$$

Pour chaque m, nous construisons une dérivation π_m de

$$\mathbf{W}(x), \mathbf{W}(c_0), \mathbf{W}(c_1), \mathbf{W}(c_2) \vdash \forall z.\mathbf{W}(z) \rightarrow \wedge_{i=0,1,2}\mathbf{W}(\mathtt{L}^i_m(z, x, c_0, c_1, c_2))$$

dans laquelle c_0, c_1 et c_2 sont des variables qui peuvent être remplacées par n'importe quel terme, à la différence de x.

Pour $m = 0$, nous avons la dérivation π_0

$$\dfrac{\begin{array}{c}\vdots\ \pi_{\mathtt{D}_i} \\ \forall z.\mathbf{W}(z) \rightarrow \wedge_{i=0,1,2}\mathbf{W}(\mathtt{D}_i(c_0, c_1, c_2))\end{array}}{\forall z.\mathbf{W}(z) \rightarrow \wedge_{i=0,1,2}\mathbf{W}(\mathtt{L}^i_m(z, x, c_0, c_1, c_2))}\ \mathrm{R}$$

Supposons que la dérivation π_m soit construite. La dérivation π_{m+1} est écrite dans la figure 6.11.

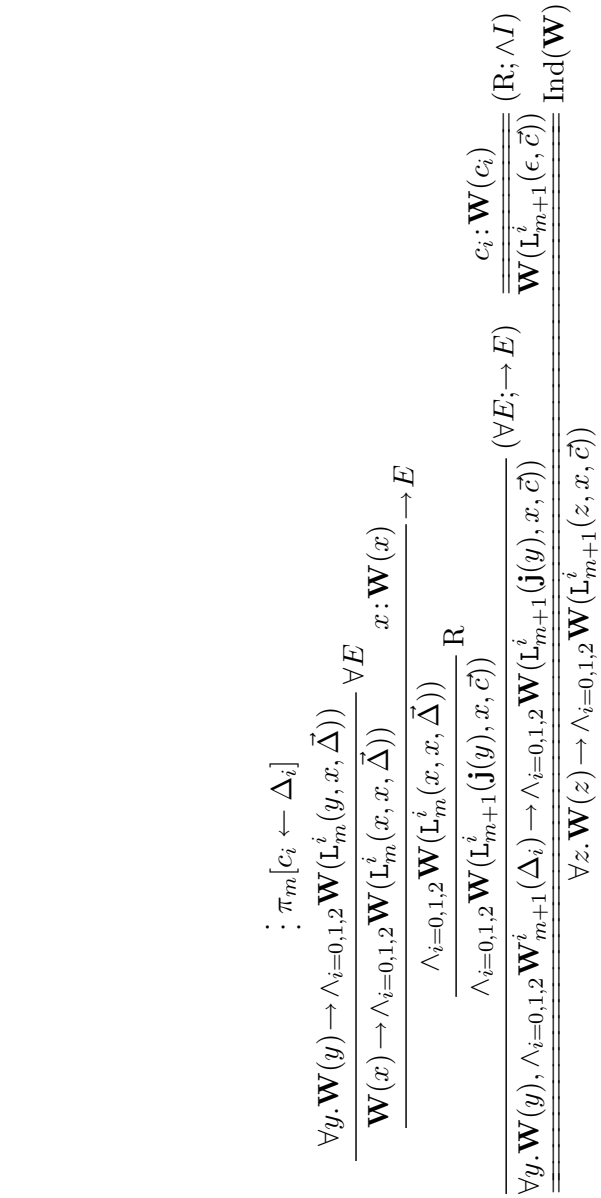

FIG. 6.11 – dérivation π_{m+1}.

Nous voyons qu'il est nécessaire de ne pas avoir de restriction sur les variables c_0, c_1, c_2 et que d'autre part, la prémisse $\mathbf{W}(x)$ est un prédicat de récurrence. Enfin, la dérivation de Tot`f` est construit ainsi

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{v:\mathbf{W}(v)\}}{\vdots\,\pi_{c+1}}}{\dfrac{\forall z.\mathbf{W}(z)\to\wedge_{i=0,1,2}\mathbf{W}(\mathtt{L}^i_{c+1}(z,v,q_0,\boldsymbol{\epsilon},v))}{\mathbf{W}(c)\to\wedge_{i=0,1,2}\mathbf{W}(\mathtt{L}^i_{c+1}(c,v,q_0,\boldsymbol{\epsilon},v))}\,\forall E} \qquad \dfrac{\vdots\,(c\in\mathcal{T}(\mathbf{word}))}{\mathbf{W}(c)}}{\dfrac{\mathbf{W}(\mathtt{L}^2_{c+1}(c,v,q_0,\boldsymbol{\epsilon},v))}{\mathbf{W}(\mathtt{f}(v))}\,\mathrm{R}}\,(\to E;\wedge E)}{\mathrm{Tot}(\mathtt{f})\equiv\forall v.\mathbf{W}(v)\to\mathbf{W}(\mathtt{f}(v))}\,(\to I;\forall I)$$

□

Troisième partie

Applications de la complexité implicite des calculs

Chapitre 7

Ordre et terminaison

Un programme termine si son calcul s'arrête sur toutes entrées. Pour démontrer la terminaison d'un programme, une méthode consiste à définir un ordre bien fondé sur les termes, et à prouver qu'à chaque étape de réduction le terme obtenu décroît. Bien que ce petit chapitre traite d'aspects fondamentaux, il n'est pas essentiel à la bonne compréhension de la suite. Nous verrons comment Dershowitz a énoncé, à partir du théorème de Higman-Kruskal, des conditions suffisantes pour qu'un ordre sur les termes démontre la terminaison d'un programme. La plupart des ordres utilisés dans la pratique vérifient ces propriétés. Ces ordres sont appelés ordres de simplification et nous donnerons des exemples dans le chapitre suivant. Notre guide pour ce chapitre a été l'état de l'art de Gallier [Gal91] et celui de Dershowitz [Der87].

7.1 Pré-ordre

Définition 7.1. Un *pré-ordre* $\preccurlyeq$ est une relation binaire réflexive et transitive sur $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$.

Les relations d'équivalences et les relations d'ordres sont des pré-ordres particuliers. A partir d'un pré-ordre $\preccurlyeq$, nous obtenons une relation d'équivalence $\approx$ définie par $t \approx s$ ssi $t \preccurlyeq s$ et $s \preccurlyeq t$. De même, un ordre partiel strict $\prec$ est défini par $t \prec s$ ssi $t \preccurlyeq s$ et $s \not\preccurlyeq t$. Définissons $t \succ s$ ssi $s \prec t$.

Définition 7.2. Un *beau pré-ordre* est un pré-ordre tel que pour toute suite infinie de termes $t_1, \ldots, t_i, \ldots, t_j, \ldots$, il existe i et j tels que $i < j$ et $t_i \preccurlyeq t_j$.

Un pré-ordre $\sqsubseteq$ est une extension de $\preccurlyeq$ si pour tous termes t et s tel que $t \preccurlyeq s$, nous avons $t \sqsubseteq s$

Proposition 7.3. *Toute extension d'un beau pré-ordre est un beau pré-ordre.*

Beau pré-ordre et pré-ordre bien fondé ne doivent pas être confondus.

Définition 7.4. Un pré-ordre est *bien fondé*, si toute chaîne décroissante est finie, *i.e.* toute suite $t_1, \ldots, t_i, \ldots$ telle que $t_i \succ t_{i+1}$ est finie. Dans ce cas, nous dirons que $\succ$ est *noethérien*.

En fait, la notion de beau pré-ordre est plus forte que la notion de bonne fondation.

Théorème 7.5. *Les assertions suivantes sont équivalentes*

1. *$\preccurlyeq$ est un beau pré-ordre.*
2. *Il n'y a pas de chaîne décroissante infinie, ni d'antichaîne infinie.*
3. *Tout pré-ordre qui étend $\preccurlyeq$ est bien fondé.*
4. *Toute suite infinie contient une chaîne infinie croissante.*

7.2 Terminaison des programmes

Définition 7.6. Un programme $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ termine lorsque la relation de réduction $\xrightarrow{*}$, associée aux règles $\mathcal{E}$, est noethérienne.

Autrement dit, en partant d'un terme quelconque, nous aboutissons en un nombre fini d'étapes de réductions à une forme normale.

Un *ordre de réduction* $\prec$ est un ordre partiel bien fondé qui est monotone et stable par substitution.

(Monotonie)

$$t \prec s \text{ implique } u[t] \prec u[t \leftarrow s]$$

(Stable par substitution)

$$\text{Pour chaque substitution } \sigma,\ t \preccurlyeq s \text{ implique } t\sigma \preccurlyeq s\sigma$$

Théorème 7.7. *Soit $\prec$ un ordre de réduction. Si pour chaque règle $l \to r \in \mathcal{E}$, nous avons $l \succ r$, alors $\mathtt{f}$ termine.*

7.3 Théorème de Higman-Kruskal

Soit $\preccurlyeq$ un pré-ordre sur les termes. Le *pré-ordre de plongement* $\sqsubseteq$ est le plus petit pré-ordre sur les termes tel que $t = \mathtt{f}(t_1, \ldots, t_n) \sqsubseteq s = \mathtt{g}(s_1, \ldots, s_m)$ si

soit $t \preccurlyeq s$ et $t_k \sqsubseteq s_{j_k}$ pour tout $1 \leq k \leq n$ et $1 \leq j_1 < \ldots < j_n \leq n$.
sinon $\mathtt{f}(t_1, \ldots, t_n) \sqsubseteq s_i$, pour $1 \leq i \leq m$.

Nous énonçons une généralisation, que l'on trouve dans l'article de Gallier [Gal91], du théorème de Higman-Kruskal [Hig52, Kru60].

Théorème 7.8. *$\preccurlyeq$ est un beau pré-ordre si et seulement si $\sqsubseteq$ est un beau pré-ordre.*

Remarque 7.9. La démonstration du thèorème est écrite dans l'article de Gallier [Gal91]. Une démonstration courte, mais d'une version plus faible, est dans l'article de Dershowitz [Der87]. Enfin, nous renvoyons le lecteur qui n'apprécie que les méthodes constructives, à l'article de Cichon et Tahham Bittar [CB98] ou à l'article de Coquand et Fridlender [CF93].

7.4 Ordre de simplification

Un *ordre de simplification* $\prec$ est un ordre partiel qui est monotone et qui possède la propriété du sous-terme

(sous-terme) $t \prec \mathtt{f}(\ldots, t, \ldots)$

Les ordres de simplification et les résultats ci-dessous sont dus à Dershowitz [Der82].

Théorème 7.10. *Soit $\mathcal{X}_n$ un ensemble de n variables. Un ordre de simplification $\prec$ sur $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}_n)$ est bien fondé.*

Démonstration. Définissons $\preccurlyeq$ par $t \preccurlyeq s$ ssi le symbole de tête de t est identique à celui de s. $\preccurlyeq$ est une relation d'équivalence et donc un beau pré-ordre, car la signature est finie. Le pré-ordre de plongement $\sqsubseteq$, engendré à partir de $\preccurlyeq$, est un beau pré-ordre d'après le théorème de Higman-Kruskal. Il est possible de montrer que $\prec$ est une extension de $\sqsubset$. Par la proposition 7.3 $\prec$ est un beau pré-ordre. Suivant le théorème 7.5, nous concluons que $\prec$ est bien fondé. □

Théorème 7.11. *Soit $\mathfrak{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ un programme et $\prec$ un ordre de simplification sur $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$ qui est stable par substitution. Si pour chaque règle $l \to r \in \mathcal{E}$, nous avons $l \succ r$, alors $\mathfrak{f}$ termine.*

Démonstration. Soit t un terme de $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$. Puisque pour chaque règle $l \to r \in \mathcal{E}$, on a $var(r) \subset var(l)$. De ce fait, les variables qui apparaissent dans une dérivation qui commence par t, sont celles de t. Nous pouvons appliquer les théorèmes 7.10 et 7.7. □

Chapitre 8

Expressivité et ordres de simplification

Au chapitre précédent, nous avons défini une classe d'ordres, les ordres de simplification, dont les propriétés étaient suffisantes pour montrer la terminaison d'un programme.

Ce chapitre présente et examine l'expressivité d'ordres de simplification bien connus et utilisés. Nous entendons par expressivité d'un ordre, la complexité des programmes de `E` qui terminent par cet ordre. Nous verrons que dans le cas des preuves de terminaison par interprétation polynomiale, les programmes sont effectivement calculables. Plus exactement, nous caractérisons les classes PTIME, DTIME($2^{O(n)}$) et DTIME($2^{2^{O(n)}}$). En conséquence, les preuves par interprétation polynomiale peuvent déterminer la complexité pratique d'un programme. D'un autre côté, l'expressivité des ordres récursifs sur les chemins est énorme. Nous capturons les fonctions primitives récursives et les fonctions multiples récursives, suivant la manière de comparer les suites d'arguments. L'intérêt de ces ordres est pleinement justifié par le fait qu'ils capturent des schémas d'algorithmes utiles dans la pratique.

8.1 Terminaison par interprétation polynomiale

Le premier exemple de preuve de terminaison par un ordre de simplification consiste à interpréter les termes sur une structure algébrique munie d'un ordre bien fondé. Cette approche, dite de terminaison par interprétation (ou sémantique), débute avec les travaux de Manna et Ness [MN70]. Lankford [Lan79] s'est intéressé aux preuves de terminaison par interprétation polynomiale, que nous allons maintenant décrire.

Définition 8.1. Une interprétation $[\ ]$ associe à chaque symbole f d'arité n, de $\mathcal{C} \cup \mathcal{F}$ une fonction $[f] : \mathbb{N}^n \to \mathbb{N}$ qui vérifie

1. $[f]$ est croissante suivant chaque argument,
   i.e. pour tout $1 \leq i \leq n$

$$X_i \leq Y_i \text{ implique } [f](\ldots, X_i, \ldots) \leq [f](\ldots, Y_i, \ldots)$$

2. $[f](\ldots, X, \ldots) > X$.

Un terme clos $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{C}, \mathcal{F})$ est interprété par

$$[f(t_1, \ldots, t_n)] = [f]([t_1], \ldots, [t_n])$$

Définition 8.2. Un programme admet une interprétation si pour toute équation $l \to r$, et pour toute substitution close σ, on a $[r\sigma] < [l\sigma]$.

Proposition 8.3. *Un programme qui admet une interprétation termine.*

Démonstration. Définissons l'ordre suivant sur les termes de $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) : s \prec t$ ssi pour toute substitution close σ, $[s\sigma] < [t\sigma]$. Nous voyons que $\preccurlyeq$ est un ordre de simplification, stable par substitution. D'après le théorème 7.11, le programme termine. □

Définition 8.4. Une interprétation [ ] d'un programme est une *interprétation polynomiale* si, pour chaque symbole f d'arité n de $\mathcal{C} \cup \mathcal{F}$, $[f]$ est un polynôme non-nul à n indéterminées et à coefficients entiers.

Nous déduisons de la proposition précédente que tout programme qui admet une interprétation polynomiale, termine.

8.1.1 Expressivités

Les travaux de Lautemann [Lau88, HL88] ont montré que la longueur d'une dérivation d'un système de réécriture qui termine par une interprétation polynomiale est bornée par une double exponentielle dans la taille du terme de départ. Dans le cadre de l'étude de la terminaison des programmes, la situation s'éclaircit, car nous distinguons les constructeurs, des symboles de fonction. C'est l'idée de Cichon et Lescanne dans l'article [CL92]. Sur le domaine des entiers bâtons, ils ont établi que la croissance des fonctions ainsi définies était polynomiale. Par la suite, Bonfante, Cichon, Marion et Touzet [BCMT99, BCMT00] ont proposé trois classes d'interprétations des constructeurs.

Classe 0 : Polynômes de la forme $P(X_1, \ldots, X_n) = \sum_{i=1}^{n} X_i + \gamma$,
où $\gamma > 0$.

Classe 1 : Polynômes de la forme $P(X_1, \ldots, X_n) = \sum_{i=1}^{n} \alpha_i X_i + \gamma$,
où au moins un α_i est > 1.

Classe 2 : Polynômes de la forme $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} X_i^{\beta_i} + R(X_1, \ldots, X_n)$,
où au moins un β_i est > 1, et R est un polynôme quelconque.

Définition 8.5. La classe des programmes $\Pi(i)$, pour $i = 0, 1, 2$, est la classe des programmes qui admettent une interprétation polynomiale pour laquelle les constructeurs sont interprétés par un polynôme d'une classe $\leq i$.

Exemple 8.6. Nous illustrons la relation entre l'interprétation d'un constructeur et sa capacité d'itération. A droite, nous avons indiqué une interprétation polynomiale possible, qui assure la terminaison. Cependant, démontrer qu'un programme termine par une interprétation polynomiale est indécidable. De ce

fait, il n'existe que des heuristiques pour trouver le bon polynôme. Des heuristiques sont décrites dans les rapports sur les implantations de système par Giesl [Gie95] ou par Ben Cherifa et Lescanne [CL87].

$[\![\mathtt{add}]\!](n,m) = n+m$

$$\begin{array}{rll} \mathtt{add}(x,\mathbf{0}) \to x & & [\mathbf{0}] = 1 \\ \mathtt{add}(\mathbf{0},y) \to y & & [\mathbf{suc}_0](X) = X+2 \\ \mathtt{add}(\mathbf{suc}_0(x),\mathbf{suc}_0(y)) \to \mathbf{suc}_0(\mathbf{suc}_0(\mathtt{add}(x,y))) & & [\mathtt{add}](X,Y) = 2X+Y+1 \end{array}$$

et $[\![\mathtt{mul}]\!](n,m) = n \cdot m$

$$\begin{array}{rll} \mathtt{mul}(\mathbf{0},y) \to \mathbf{0} & & \\ \mathtt{mul}(\mathbf{suc}_0(x),y) \to \mathtt{add}(y,\mathtt{mul}(x,y)) & & [\mathtt{mul}](X,Y) = (X+1)(Y+1) \end{array}$$

$[\![\mathtt{exp}]\!](n) = 2^n$

$$\begin{array}{rll} \mathtt{exp}(\mathbf{0}) \to \mathbf{suc}_0(\mathbf{0}) & & [\mathbf{suc}_1](X) = 4 \cdot (X+4) \\ \mathtt{exp}(\mathbf{suc}_1(x)) \to \mathtt{add}(\mathtt{exp}(x),\mathtt{exp}(x)) & & [\mathtt{exp}](X) = X+3 \end{array}$$

$[\![\mathtt{exp}_2]\!](n) = 2^{2^n}$

$$\begin{array}{rll} \mathtt{exp}_2(\mathbf{0}) \to \mathbf{suc}_0(\mathbf{suc}_0(\mathbf{0})) & & [\mathbf{suc}_2](X) = (X+6)^4 \\ \mathtt{exp}_2(\mathbf{suc}_2(x)) \to \mathtt{mul}(\mathtt{exp}_2(x),\mathtt{exp}_2(x)) & & [\mathtt{exp}_2](X) = X+5 \end{array}$$

Par exemple, pour la troisième équation de `add`, nous avons

$$\begin{aligned} [\mathtt{add}(\mathbf{suc}_0(x),\mathbf{suc}_0(y))] &= 2(X+2)+(Y+2)+1 \\ &> [\mathbf{suc}_0(\mathbf{suc}_0(\mathtt{add}(x,y)))] = ((2X+Y+1)+2)+2 \end{aligned}$$

Les programmes `add` et `mul` appartiennent à $\Pi(0)$, car ils sont définis sur les constructeurs $\mathbf{0}$ et $\mathbf{suc}_0$, dont l'interprétation est de classe 0. Le programme `exp` de l'exponentiel nécessite un autre constructeur $\mathbf{suc}_1$ dont l'interprétation est de classe 1. De ce fait, `exp` appartient à $\Pi(1)$. Enfin, $\mathtt{exp}_2$ est dans $\Pi(2)$. En fait, nous travaillons sur trois structures $\langle \mathbf{nat}_i, \mathbf{0}, \mathbf{suc}_i \rangle$, ce qui ressemble à la ramification induite par les valences, du chapitre 4. Il semble possible de donner une interprétation algébrique de la notion d'itération ramifiée. Pour cela, les constructeurs seraient interprétés par des polynômes de la classe 0.

Théorème 8.7.

1. $\mathcal{F}(\Pi(0))$ *est exactement la classe* PTIME *des fonctions calculables en temps polynomial.*
2. $\mathcal{F}(\Pi(1))$ *est exactement la classe* $\mathrm{DTIME}(2^{O(n)})$ *des fonctions calculables en temps exponentiel.*
3. $\mathcal{F}(\Pi(2))$ *est exactement la classe* $\mathrm{DTIME}(2^{2^{O(n)}})$ *des fonctions calculables en temps doublement exponentiel.*

Un attrait de ce résultat est que la preuve de terminaison fournit la complexité du programme. Dans le cas des interprétations exponentielles [Les92], nous avons montré [BCMT00] que nous pouvions caractériser les fonctions super-élémentaires. Nous pourrions envisager des interprétations à croissance sous-polynomiale afin d'analyser plus précisément la complexité d'un programme et aussi d'augmenter la classe des programmes qui terminent.

Remarque 8.8. A la suite, Bonfante [Bon00] a étudié dans sa thèse des restrictions de l'ordre KBO (*Knuth Bendix Ordering*) qui délimitent, entre autre, des classes de fonctions de complexité bornée en espace.

8.1.2 Non-déterminisme

Avec un programme non-confluent, un terme a plusieurs formes normales. L'évaluation d'un programme non-confluent peut être vue comme un mécanisme non-déterministe, dans lequel il faut choisir une réduction, à chaque pas.

Exemple 8.9. Considérons le problème qui consiste à trouver une clique maximal dans un graphe. Chaque sommet du graphe est codé par un mot de $\langle \mathbf{word},\ \boldsymbol{\epsilon}:\mathbf{word},\ \mathbf{0}:\mathbf{word},\ \mathbf{1}:\mathbf{word}\rangle$. Nous disposons de la liste V des sommets, et de la liste E des arcs. La liste E est codée par $E = (u_0 * v_0 * u_1 * v_1 * \ldots)$ et signifie qu'il y a un arc entre u_0 et v_0, un arc entre u_1 et v_1, ... Le constructeur $*$ est la notation infixée de **cons**.

$$\begin{array}{rrr}
\texttt{and}(\mathbf{tt},\mathbf{tt}) \to \mathbf{tt} & \texttt{and}(\mathbf{ff},x) \to \mathbf{ff} & \texttt{and}(x,\mathbf{ff}) \to \mathbf{ff} \\
\texttt{eq?}(\boldsymbol{\epsilon},\boldsymbol{\epsilon}) \to \mathbf{tt} & \texttt{eq?}(\mathbf{i}(u),\boldsymbol{\epsilon}) \to \mathbf{ff} & \texttt{eq?}(\boldsymbol{\epsilon},\mathbf{j}(v)) \to \mathbf{ff} \\
\texttt{eq?}(\mathbf{i}(u),\mathbf{i}(v)) \to \texttt{eq?}(u,v) & \texttt{eq?}(\mathbf{i}(u),\mathbf{j}(v)) \to \mathbf{ff} &
\end{array}$$

Le prédicat `find` qui teste si un couple de sommet est un arc de G :

$$\begin{array}{l}
\texttt{if}\ \mathbf{tt}\ \texttt{then}\ x\ \texttt{else}\ y \to x \\
\texttt{if}\ \mathbf{ff}\ \texttt{then}\ x\ \texttt{else}\ y \to y
\end{array}$$

$$\begin{array}{r}
\texttt{find}(u,v,\mathbf{nil}) \to \mathbf{ff} \\
\texttt{find}(u,v,u' * v' * E) \to \texttt{if}\ \texttt{and}(\texttt{eq?}(u,u'),\texttt{eq?}(v,v'))\ \texttt{then}\ \mathbf{tt}\ \texttt{else}\ \texttt{find}(u,v,E)
\end{array}$$

Le programme $\texttt{clique}(K,E)$ retourne **tt** si l'ensemble des sommets de K forme un sous graphe complet de G. Pour cela, `clique` utilise $\texttt{complete}(u,V,E)$ qui vérifie que le sommet u est relié à tous les sommets de V en utilisant des arcs de E.

$$\begin{array}{r}
\texttt{complete}(u,\mathbf{nil},E) \to \mathbf{tt} \\
\texttt{complete}(u,v * V,E) \to \texttt{if}\ \texttt{find}(u,v,E)\ \texttt{then}\ \texttt{complete}(u,V,E)\ \texttt{else}\ \mathbf{ff} \\
\texttt{clique}(\mathbf{nil},E) \to \mathbf{tt} \\
\texttt{clique}(u * K,E) \to \texttt{if}\ \texttt{complete}(u,K,E)\ \texttt{then}\ \texttt{clique}(K,E)\ \texttt{else}\ \mathbf{ff}
\end{array}$$

L'ensemble des formes normales de $\texttt{choice}(V, \mathbf{nil}, E)$ est l'ensemble de toutes les cliques de G. Ainsi, $[\![\texttt{choice}(V, \mathbf{nil}, E)]\!]$ retourne la clique maximale.

$$\begin{aligned}
\texttt{choice}(\mathbf{nil}, K, E) &\to \texttt{if clique}(K, E) \texttt{ then } K \texttt{ else } \mathbf{nil} \\
\texttt{choice}(u * V, K, E) &\to \texttt{choice}(V, u * K, E) \\
\texttt{choice}(u * V, K, E) &\to \texttt{choice}(V, K, E)
\end{aligned}$$

Les deux dernières équations ne sont pas confluentes et correspondent à un choix non-déterministe.

Le lecteur courageux pourra vérifier que le programme termine à partir des interprétations polynomiales suivantes :

$$\begin{aligned}
[\mathbf{ff}] = [\mathbf{tt}] = [\boldsymbol{\epsilon}] = [\mathbf{nil}] &= 1 \\
[\mathbf{i}](X) &= X + 2 \\
[*](X, Y) &= X + Y + 4 \\
[\texttt{eq?}](X, Y) = [\texttt{and}](X, Y) &= X + Y + 1 \\
[\texttt{if}](X, Y, Z) &= X + Y + Z + 1 \\
[\texttt{find}](X, Y, Z) &= (X + Y + 1)(Z + 1) \\
[\texttt{complete}](X, Y, Z) &= (X + 1)(Y + 1)(Z + 1) \\
[\texttt{clique}](X, Y) &= (X + 1)^2 \cdot (Y + 1) \\
[\texttt{choice}](X, Y, Z) &= (2X + Y + 1)^2 \cdot (Z + 1)
\end{aligned}$$

Pour donner un sens fonctionnel à un calcul non-déterministe, nous suivons l'idée de Krentel [Kre88], repris par Grädel et Gurevich dans [GG95].

Définition 8.10. Supposons que $\texttt{f}$ soit de type $\mathbf{s}_1, \ldots, \mathbf{s}_n \to \mathbf{s}$. Le programme (non nécessairement confluent) $\texttt{f}$ est interprété par la fonction $[\![\texttt{f}]\!] : [\![\mathbf{s}_1]\!] \times \ldots [\![\mathbf{s}_n]\!] \to [\![\mathbf{s}]\!]$ qui est définie ainsi :

$$[\![\texttt{f}]\!](\mathbf{a}_1, \ldots, \mathbf{a}_n) = \max\{\mathbf{d} : \texttt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n) \xrightarrow{!} \mathbf{d}\}$$

où le maximum est pris suivant un ordre bien fondé sur $[\![\mathbf{s}]\!]$.

La fonction d'horloge décrite en 2.4, s'étend naturellement à la nouvelle sémantique.

$$\mathtt{E}^{Ntime(T(n))} = \{\texttt{f} : \forall \vec{\mathbf{a}} \in \mathcal{T}(\mathcal{C}), \mathit{Clock}_{\texttt{f}}(\vec{\mathbf{a}}) \leq T(n), \text{ où } n = \sum_{1 \leq i \leq n} |\mathbf{a}_i|\}$$

Ensuite, la classe des fonctions calculables en temps non-déterministe $T(n)$ est définie par

$$\mathrm{NDTIME}(T(n)) = \{[\![\texttt{f}]\!] : \texttt{f} \in \mathtt{E}^{Ntime(T(n))}\}$$

Enfin, la classe des fonctions calculables en temps polynomial est

$$\mathrm{NPTIME} = \bigcup_{T \text{ est un polynôme}} \mathrm{NDTIME}(T(n))$$

Définissons $\Delta(i)$ comme l'ensemble des programmes non nécessairement confluents qui admettent une interprétation polynomiale pour laquelle les constructeurs sont interprétés par un polynôme d'une classe $\leq i$, pour $i = 0, 1, 2$.

Théorème 8.11.

1. $\mathcal{F}(\Delta(0))$ *est exactement la classe* NPTIME.
2. $\mathcal{F}(\Delta(1))$ *est exactement la classe* NDTIME$(2^{O(n)})$.
3. $\mathcal{F}(\Delta(2))$ *est exactement la classe* NDTIME$(2^{2^{O(n)}})$.

8.2 Ordre récursif sur les chemins avec statuts

8.2.1 Extensions d'un ordre aux suites

Soit $\preccurlyeq$ un pré-ordre sur $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$. Nous décrivons trois extensions possibles de $\preccurlyeq$ aux suites finies de termes de même longueur.

Multi-Ensemble

Commençons par l'extension la plus connue. Un *multi-ensemble* est une structure dans laquelle les éléments peuvent être répétés plusieurs fois.

Définition 8.12. Le pré-ordre multi-ensemble $\preccurlyeq^{\mathrm{m}}$ engendré par $\preccurlyeq$, est défini par $(s_1, \ldots, s_n) \preccurlyeq^{\mathrm{m}} (t_1, \ldots, t_n)$ ssi $\{s_1, \ldots, s_n\} \preccurlyeq^{\mathrm{m}} \{t_1, \ldots, t_n\}$ en suivant les règles ci-dessous.

$$\frac{}{\{t_1, \ldots, t_m\} \preccurlyeq^{\mathrm{m}} \{t_1, \ldots, t_m\}}$$

$$\frac{s_i \approx t_j \qquad \{s_1, \ldots, s_n\} \setminus \{s_i\} \preccurlyeq^{\mathrm{m}} \{t_1, \ldots, t_m\} \setminus \{t_j\}}{\{s_1, \ldots, s_n\} \preccurlyeq^{\mathrm{m}} \{t_1, \ldots, t_m\}}$$

$$\frac{s_{i_1} \prec t_j \ldots s_{i_k} \prec t_j \qquad \{s_1, \ldots, s_n\} \setminus \{s_{i_1}, \ldots, s_{i_k}\} \preccurlyeq^{\mathrm{m}} \{t_1, \ldots, t_m\} \setminus \{t_j\}}{\{s_1, \ldots, s_n\} \preccurlyeq^{\mathrm{m}} \{t_1, \ldots, t_m\}}$$

Produit

Nous abandonnerons le pré-ordre multi-ensemble par la suite, au profit de l'ordre produit qui en est une restriction.

Définition 8.13. L'ordre produit $\prec^{\mathrm{p}}$ engendré par $\preccurlyeq$, est défini par la règle *(Produit)* de la figure 8.1. Posons $\preccurlyeq^{\mathrm{p}} = \prec^{\mathrm{p}} \cup =$.

$$\textit{(Produit)} \quad \frac{s_1 \preccurlyeq t_1 \ldots s_i \prec t_i \ldots s_n \preccurlyeq t_n}{(s_1, \ldots, s_n) \prec^{\mathrm{p}} (t_1, \ldots, t_n)}$$

FIG. 8.1 – Extension produit.

Lexicographique

Définition 8.14. L'ordre lexicographique $\prec^{\ell}$ engendré par $\preccurlyeq$, est défini par la règle *(Lexicographique)* de la figure 8.2. Posons $\preccurlyeq^{\ell} = \prec^{\ell} \cup =$.

$$\textit{(Lexicographique)} \quad \frac{\ldots t_i \approx s_i \ldots t_k \prec s_k \ldots}{(s_1, \ldots, s_n) \prec^{\ell} (t_1, \ldots, t_n)}$$

FIG. 8.2 – Extension lexicographique.

Remarque 8.15. Si $\prec$ est un ordre, les extensions produit et lexicographique sont des ordres, mais ce n'est pas le cas de l'extension multi-ensemble qui reste un pré-ordre.

8.2.2 Ordre récursif avec statut sur les chemins

Une *précédence* est un pré-ordre $\preceq_{\mathcal{F}}$ sur les symboles de fonctions $\mathcal{F}$ tel que si $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ alors $\mathtt{g}$ et $\mathtt{f}$ ont la même arité. A chaque symbole de fonction est associé un statut. Celui-ci indique comment comparer les arguments de deux termes qui ont des symboles de tête équivalents.

Définition 8.16. Un *statut* τ est une fonction de $\mathcal{F}$ dans $\{m, p, \ell\}$ qui est compatible avec $\preceq_{\mathcal{F}}$, *i.e.* si $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ alors $\tau(\mathtt{g}) = \tau(\mathtt{f})$.

Nous distinguons trois sortes de comparaisons possibles : $\prec^{\mathrm{m}}$ pour l'extension multi-ensemble, $\prec^{\mathrm{p}}$ pour l'extension produit, et $\prec^{\ell}$ pour l'extension lexicographique.

Définition 8.17. Les relations $\prec_{rpo}$ et $\preceq_{rpo}$ sont définies par les règles de la figure 8.3.

$$\textit{(Id)} \quad \frac{}{t \preceq_{rpo} t}$$

$$\textit{(Prj)} \quad \frac{s \preceq_{rpo} t_i}{s \prec_{rpo} f(\ldots, t_i, \ldots)}\ f \in \mathcal{C} \cup \mathcal{F}$$

$$\textit{(Fnc)} \quad \frac{s_1 \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n) \quad \ldots \quad s_m \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}{\mathtt{g}(s_1, \ldots, s_m) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}\ \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$$

$$\textit{(Cns)} \quad \frac{s_1 \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n) \quad \ldots \quad s_m \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}{\mathbf{c}(s_1, \ldots, s_m) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}\ \mathbf{c} \in \mathcal{C}$$

$$\textit{(Sta)} \quad \frac{(s_1, \ldots, s_n) \preceq_{rpo}^{\tau(\mathtt{f})} (t_1, \ldots, t_n) \quad \forall i \leq n, \quad s_i \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}{\mathtt{g}(s_1, \ldots, s_n) \preceq_{rpo} \mathtt{f}(t_1, \ldots, t_n)}\ \mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$$

FIG. 8.3 – RPO : définition de $\preceq_{rpo}$ et $\prec_{rpo}$.

Proposition 8.18. *La relation $\preceq_{rpo}$ est un ordre de simplification.*

Un programme $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ est ordonné par RPO (*Recursive Path Ordering with statuts*) s'il existe une précédence $\prec_{\mathcal{F}}$ et un statut τ tels que pour chaque équation $l \to r \in \mathcal{E}$, nous avons $r \prec_{rpo} l$.

Proposition 8.19. *Tout programme ordonné par* RPO *termine.*

Remarque 8.20. La présentation de RPO tient compte de la distinction faite entre les constructeurs et les symboles de fonctions. Les constructeurs sont considérés comme les éléments minimaux pour la précédence $\preceq_{\mathcal{F}}$. Les variables et les constructeurs sont incomparables.

8.2.3 Expressivités

Lorsque le statut est limité, nous retrouvons des ordres bien connus. Quand tous les symboles de fonction ont un statut multi-ensemble, *i.e.* $\forall \mathtt{f} \in \mathcal{F},\ \tau(\mathtt{f}) = m$, l'ordre défini correspond exactement avec l'ordre multi-ensemble sur les chemins MPO (*Multiset Path Orderings*). MPO a été défini par Dershowitz [Der82]. Plaisted [Pla78] avait proposé un ordre semblable. Rusinowitch [Rus87] a montré leur équivalence, sous certaines conditions. Cichon et Hofbauer [Hof92] ont caractérisé, indépendamment, l'expressivité de MPO.

Théorème 8.21. *La classe des fonctions calculées par un programme ordonné par* MPO *est exactement la classe des fonctions primitives récursives.*

Dans le cas où tous les symboles de fonction ont un statut produit, *i.e.* $\forall \mathtt{f} \in \mathcal{F},\ \tau(\mathtt{f}) = p$, l'expressivité reste identique à MPO. Ce résultat est un corollaire évident du précédent théorème.

Enfin, si tous les symboles de fonction ont un statut lexicographique, *i.e.* $\forall \mathtt{f} \in \mathcal{F},\ \tau(\mathtt{f}) = \ell$, alors l'ordre défini est l'ordre lexicographique sur les chemins, introduit par Kamin et Lévy [KL80], que nous abrégerons par LPO (*Lexicographic Path Orderings*). Weiermann [Wei95] a établi le pouvoir expressif de LPO

Théorème 8.22. *La classe des fonctions calculées par un programme ordonné par* LPO *est exactement la classe des fonctions multiples récursives.*

Chapitre 9

Vers une analyse des algorithmes

Nous sommes arrivés au centre de la problématique de l'introduction qui est l'analyse des algorithmes. Nous montrons comment les principes qui régissent l'analyse prédicative et les ordres de terminaison, permettent de déterminer la complexité d'une fonction à partir d'un programme, *i.e.* la complexité implicite de la fonction. Une conséquence est la possibilité de transformer le programme en un programme plus efficace. Nous verrons que dans certains cas, le programme avec sa sémantique usuelle se calcule en temps exponentiel, alors que la fonction qu'il dénote est calculable en temps polynomial, par un autre algorithme.

Dans la première partie, nous redéfinirons la notion de valence, et définirons un ordre qui est une restriction de l'ordre récursif sur les chemins avec les statuts lexicographique et produit. Nous obtiendrons une caractérisation des fonctions calculables en temps polynomial qui capture des algorithmes non-triviaux.

Dans la seconde partie, nous utilisons un argument sémantique pour agrandir la classe des algorithmes qui représentent des fonctions calculables en temps polynomial.

Pour terminer, nous discuterons d'un interpréteur pour évaluer efficacement les programmes que nous avons considérés. Cet interpréteur mémorise les appels récursifs, dans un cache, pour accélérer le calcul. Nous indiquerons comment optimiser le cache pour que ses informations soient minimales.

9.1 Ordre et complexité algorithmique

9.1.1 Terminaison par restriction

Notre intention est de comprendre quels principes conduisent à la définition de pré-ordres qui (i) sont décidables et (ii) délimitent une classe de programmes calculant des fonctions de faible complexité.

L'idée de départ est simple. Considérons une relation binaire $\prec$ sur les termes.

Fait 9.1. Soit $\sqsubset$ un ordre de réduction qui est une extension de $\prec$. Si pour

chaque règle $l \to r$ d'un programme, nous avons $l \succ r$, alors le programme termine.

Démonstration. Puisque $\sqsubset$ est une extension de $\prec$, nous avons $l \sqsupset r$. Donc le programme termine car $\sqsubset$ est un ordre de réduction. □

Nous avons construit dans l'article de Marion [Mar00] un ordre partiel qui est une restriction de MPO et qui capture PTIME. Tandis que Cichon et Marion [CM00] ont défini un ordre partiel qui est une restriction de LPO et qui caractérise PSPACE. Ces ordres ne sont pas des ordres de réductions. Ils ne sont pas monotones, et de ce fait, deux termes d'une dérivation peuvent être incomparables.

9.1.2 Valence

Nous redéfinissons la notion de valence, vue au chapitre 4, pour sortir du cadre strict de l'analyse prédicative.

Définition 9.2. Une *valence* est une fonction ν qui associe à chaque symbole de fonction $\mathtt{f}$ d'arité n et à chaque position d'argument i,$1 \leq i \leq n$ une valeur de $\{0, 1\}$. Nous écrirons $\nu(\mathtt{f}, i)$ pour indiquer la valence de l'argument i, $1 \leq i \leq n$.

La valence d'un symbole de fonction indique le comportement algorithmique des arguments. Le paramètre de récurrence sera de valence 1, et les paremètres des arguments critiques seront de valence 0.

Dans une preuve de terminaison, la valence sert à préciser comment les arguments sont comparés. Pour cela, nous définirons deux ordres $\prec_1$ et $\prec_0$ pour comparer les arguments de valence 1 et ceux de valence 0. Les valences généralisent les statuts définis au chapitre 8, pour comparer les arguments en fonction du sysmbole de tête. Nous ne distinguons pas seulement les symboles de tête d'un terme, mais aussi les arguments.

Notation 9.3. Pour simplifier l'écriture, nous emploierons la convention de Bellantoni et Cook. Nous séparons les arguments de valence 1 (normaux) de ceux de valence 0 (sûrs) par un point-virgule. Ainsi, $\mathtt{f}(\vec{t}; \vec{u})$ signifie $\nu(\mathtt{f}, i) = 1$ pour tout $1 \leq i \leq n$ si $\vec{t} = t_1, \ldots, t_n$ et $\nu(\mathtt{f}, j) = 0$, pour tout $n+1 \leq j \leq n+m$ si $\vec{u} = u_1, \ldots, u_m$. De plus, nous marquerons la valence sur les sortes, e.g. $\mathtt{f} : \mathbf{nat}(1), \mathbf{nat}(0) \to \mathbf{nat}$ précise que le premier argument de $\mathtt{f}$ est de valence 1, le second de valence 0, *i.e.* $\nu(\mathtt{f}, 1) = 1$ et $\nu(\mathtt{f}, 2) = 0$.

9.1.3 Ordre récursif allégé sur les chemins avec statuts

Nous dirons qu'une relation d'équivalence $\approx_{\mathcal{F}}$ respecte les valences des symboles de fonctions de $\mathcal{F}$ si $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ implique que $\mathtt{f}$ et $\mathtt{g}$ sont d'arité n et $\nu(\mathtt{f}, i) = \nu(\mathtt{g}, i)$ pour tout $1 \leq i \leq n$. Par la suite, une précédence $\preceq_{\mathcal{F}}$ respecte les valences si la relation d'équivalence associée $\approx_{\mathcal{F}}$ respecte les valences. Nous définissons $\approx$ comme la plus petite relation d'équivalence sur $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$ qui est une extension homéomorphe de $\approx_{\mathcal{F}}$, ou en plus clair

$$\frac{s_1 \approx t_1 \ldots s_n \approx t_n}{\mathtt{g}(s_1, \ldots, s_n) \approx \mathtt{f}(t_1, \ldots, t_n)}\, \mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$$

Définition 9.4. Les relations $\prec_1$ et $\prec_0$ sont définies par les règles de la figure 9.1.

$$\textit{(C-Prj)}\quad \frac{s \preceq_k t_i}{s \prec_k \mathbf{c}(\ldots, t_i, \ldots)}$$

$$\textit{(F-Prj)}\quad \frac{s \preceq_k t_i}{s \prec_k \mathtt{f}(\ldots, t_i, \ldots)} \text{ où } k \leq \nu(\mathtt{f}, i)$$

$$\textit{(Cns)}\quad \frac{\ldots s_i \prec_k \mathtt{f}(t_1, \ldots, t_m) \ldots}{\mathbf{c}(s_1, \ldots, s_n) \prec_k \mathtt{f}(t_1, \ldots, t_m)}$$

$$\textit{(Fnc)}\quad \frac{\ldots s_i \prec_1 \mathtt{f}(\vec{t}; \vec{u}) \ldots v_j \prec_0 \mathtt{f}(\vec{t}; \vec{u}) \ldots}{\mathtt{g}(\vec{s}; \vec{v}) \prec_k \mathtt{f}(\vec{t}; \vec{u})} \text{ où } \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$$

$$\textit{(Sta)}\quad \frac{(\vec{s}; \vec{v}) \prec_0^{\tau(\mathtt{f}),\mathtt{f}} (\vec{t}; \vec{u})}{\mathtt{g}(\vec{s}; \vec{v}) \prec_0 \mathtt{f}(\vec{t}; \vec{u})} \text{ où } \mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$$

FIG. 9.1 – LPRO où $\preceq_k = \prec_k \cup \approx$, $\mathtt{f} \in \mathcal{F}$, et $\mathbf{c} \in \mathcal{C}$.

$$\textit{(Prd)}\quad \frac{\ldots s_i \preceq_1 t_i \ldots s_k \prec_1 t_k \ldots s_j \preceq_1 t_j \ldots v_l \preceq_0 u_l \ldots}{(\vec{s}; \vec{v}) \prec_0^{p,\mathtt{f}} (\vec{t}; \vec{u})}$$

$$\textit{(Lex)}\quad \frac{\ldots s_i \approx t_i \ldots s_k \prec_1 t_k \ldots s_j \preceq_1 t_j \ldots v_l \preceq_0 \mathtt{f}(\vec{t}; \vec{u}) \ldots}{(\vec{s}; \vec{v}) \prec_0^{\ell,\mathtt{f}} (\vec{t}; \vec{u})} \text{ où } v_l \in \mathcal{T}(\mathcal{F}{\downarrow}\mathtt{f})$$

FIG. 9.2 – Extensions produit et lexicographique où $\mathcal{T}(\mathcal{F}{\downarrow}\mathtt{f})$ est l'ensemble des termes dont les symboles de fonction sont $\prec_{\mathcal{F}} \mathtt{f}$.

Définition 9.5. Un programme $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \Gamma \rangle$ est ordonné par LPRO (*Light Product path Recursive Ordering*) s'il existe une précédence $\preceq_{\mathcal{F}}$ et une valence ν telles que pour chaque équation $l \to r$, nous ayons $r \prec_0 l$.

Les programmes de $\mathbf{SPR}^{\omega}$, ainsi que ceux de `B`, du chapitre 4, sont ordonnés par LPRO.

Proposition 9.6. *Un programme ordonné par* LPRO *termine.*

Démonstration. $\prec_{rpo}$ est une extension de $\prec_0$ et $\prec_1$. Puisque $\prec_{rpo}$ est un ordre de réduction, nous concluons que le programme termine d'après le fait 9.1. □

Exemple 9.7. Nous allons présenter plusieurs exemples que nous avons classés par difficulté. Comme d'habitude, le dernier exemple est le plus instructif.

1. Revenons sur le programme efficace inf qui détermine le minimum de deux entiers.

   $\mathtt{inf} : \mathbf{nat}(1), \mathbf{nat}(0) \to \mathbf{nat}$

   $$\begin{aligned} \mathtt{inf}(\mathbf{0}; y) &\to \mathbf{0} \\ \mathtt{inf}(x; \mathbf{0}) &\to \mathbf{0} \\ \mathtt{inf}(\mathbf{suc}(x); \mathbf{suc}(y)) &\to \mathbf{suc}(\mathtt{inf}(x; y)) \end{aligned}$$

   Définissons la valence de $\mathtt{inf}$ par $\nu(\mathtt{inf}, 1) = 1$, $\nu(\mathtt{inf}, 2) = 0$. Les deux premières équations sont ordonnées car les deux règles de projections, *($\mathcal{C}$-Prj)* et *($\mathcal{F}$-Prj)*, impliquent que $\prec_0$ possède la propriété du sous-terme. La preuve de terminaison de la troisième règle est donnée ci-dessous.

   $$\dfrac{\dfrac{\dfrac{x \approx x}{x \prec_1 \mathbf{suc}(x)}\,(\mathcal{C}\text{-}Prj) \quad \dfrac{y \approx y}{y \prec_0 \mathbf{suc}(y)}\,(\mathcal{C}\text{-}Prj)}{\mathtt{inf}(x; y) \prec_0 \mathtt{inf}(\mathbf{suc}(x); \mathbf{suc}(y))}\,(Prd)}{\mathbf{suc}(\mathtt{inf}(x; y)) \prec_0 \mathtt{inf}(\mathbf{suc}(x); \mathbf{suc}(y))}\,(Cns)$$

   Remarquons que le programme serait aussi ordonné si $\nu(\mathtt{inf}, 2) = 1$.

2. Nous traitons le cas d'une récurrence primitive simple sur les listes d'entiers. La fonction $[\![\mathtt{Linf}]\!]$ calcule l'entier minimum d'une liste.

   $\mathtt{Linf} : \mathbf{list}(\mathbf{nat})(1) \to \mathbf{nat}$

   $$\begin{aligned} \mathtt{Linf}(\mathbf{nil};) &\to \mathbf{0} \\ \mathtt{Linf}(\mathbf{cons}(n, l);) &\to \mathtt{inf}(n; \mathtt{Linf}(l;)) \end{aligned}$$

   Posons $\nu(\mathtt{Linf}, 1) = 1$ et $\mathtt{inf} \prec_{\mathcal{F}} \mathtt{Linf}$.
   La première équation est ordonnée en appliquant *(Cns)* au cas particulier où le constructeur est d'arité 0. La preuve de terminaison de la dernière règle est la suivante.

   $$\dfrac{\dfrac{\dfrac{n \approx n}{n \prec_1 \mathbf{cons}(n, l)}\,(\mathcal{C}\text{-}Prj)}{n \prec_1 \mathtt{Linf}(\mathbf{cons}(n, l);)}\,(\mathcal{F}\text{-}Prj) \quad \dfrac{\dfrac{l \approx l}{l \prec_1 \mathbf{cons}(n, l)}\,(\mathcal{C}\text{-}Prj)}{\mathtt{Linf}(l;) \prec_0 \mathtt{Linf}(\mathbf{cons}(n, l);)}\,(Prd)}{\mathtt{inf}(n; \mathtt{Linf}(l)) \prec_0 \mathtt{Linf}(\mathbf{cons}(n, l);)}\,(Fnc)$$

3. Nous illustrons le rôle des valences pour bloquer une définition de l'exponentielle.

   $\mathtt{d} : \mathbf{nat}(1) \to \mathbf{nat}$ et $[\![\mathtt{d}]\!](\mathtt{n}) = 2 \cdot \mathtt{n}$

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

$\mathtt{exp} : \mathbf{nat} \to \mathbf{nat}$ et $[\![\mathtt{exp}]\!](\mathtt{n}) = 2^{\mathtt{n}}$

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

En posant $\nu(\mathtt{d}, 1) = 1$, nous voyons que les deux règles qui définissent $\mathtt{d}$ sont ordonnables. Remarquons que si nous avions posé $\nu(\mathtt{d}, 1) = 0$, alors nous n'aurions pu ordonner la seconde équation.

De ce fait, nous ne pouvons pas ordonner la seconde règle de $\mathtt{exp}$, parce que nous devrions montrer que $\mathtt{exp}(x) \preceq_1 \mathtt{exp}(\mathbf{suc}(x))$. Or, il est facile de voir que deux termes avec le même symbole de fonction en tête sont incomparables pour $\preceq_1$.

4. Le programme $\mathtt{renv}$ qui renverse efficacement une liste nécessite une comparaison lexicographique, à la différence des autres exemples.

$\mathtt{renv} : \mathbf{list}(\mathbf{nat})(1), \mathbf{list}(\mathbf{nat})(0) \to \mathbf{list}(\mathbf{nat})$

$$\begin{aligned} \mathtt{renv}(\mathbf{nil}; r) &\to r \\ \mathtt{renv}(\mathbf{cons}(a, l); r) &\to \mathtt{renv}(l; \mathbf{cons}(a, r)) \end{aligned}$$

$\mathtt{renverser} : \mathbf{list}(\mathbf{nat})(1) \to \mathbf{list}(\mathbf{nat})$

$$\mathtt{renverser}(x) \to \mathtt{renv}(x; \mathbf{nil})$$

Le statut de $\mathtt{renv}$ est lexicographique, *i.e.* $\tau(\mathtt{renv}) = \ell$. En posant $\mathtt{renv} \prec_{\mathcal{F}} \mathtt{renverser}$, nous voyons que chaque équation est ordonnée.

$$\dfrac{\dfrac{l \approx l}{l \prec_0 \mathbf{cons}(a, l)}\,(\mathcal{C}\text{-}Prj) \quad \dfrac{a \approx a \quad r \approx r}{\mathbf{cons}(a, r) \prec_0 \mathtt{renv}(\mathbf{cons}(a, l); r)}\,(Cns)}{\mathtt{renv}(l; \mathbf{cons}(a, r)) \prec_0 \mathtt{renv}(\mathbf{cons}(a, l); r)}\,(Lex)$$

5. Une sous-suite commune de longueur k entre deux mots $u = u_1 \dots u_m$ et $v = v_1 \dots v_n$, est décrite par deux suites d'indices $i_1 < \dots < i_k$ et $j_1 < \dots < j_k$ telles que $u_{i_q} = v_{j_q}$ pour tout $1 \leq q \leq k$. Considérons le problème qui consiste à trouver la longueur de la plus longue sous-suite commune entre deux mots de $\{0,1\}^*$ codés dans $\langle \mathbf{word},\ \boldsymbol{\epsilon} : \mathbf{word},\ \mathbf{0} : \mathbf{word},\ \mathbf{1} : \mathbf{word} \rangle$. Une solution récursive de ce problème est

$\mathtt{max} : \mathbf{word}(1), \mathbf{nat}(0), \mathbf{nat}(0) \to \mathbf{nat}$

$$\begin{aligned} \mathtt{max}(x; n, \mathbf{0}) &\to n \\ \mathtt{max}(x; \mathbf{0}, m) &\to m \\ \mathtt{max}(\mathbf{i}(x); \mathbf{suc}(n), \mathbf{suc}(m)) &\to \mathbf{suc}(\mathtt{max}(x; n, m)) \qquad \mathbf{i} \in \{\mathbf{0}, \mathbf{1}\} \end{aligned}$$

$$\texttt{lcs} : \mathbf{word}(1), \mathbf{word}(1) \to \mathbf{word}$$

$$\begin{aligned}
\texttt{lcs}(x, \boldsymbol{\epsilon};) &\to \mathbf{0} \\
\texttt{lcs}(\boldsymbol{\epsilon}, y;) &\to \mathbf{0} \\
\texttt{lcs}(\mathbf{i}(x), \mathbf{i}(y);) &\to \mathbf{suc}(\texttt{lcs}(x, y;)) \\
\texttt{lcs}(\mathbf{i}(x), \mathbf{j}(y);) &\to \texttt{max}(\mathbf{i}(x); \texttt{lcs}(x, \mathbf{j}(y);), \texttt{lcs}(\mathbf{i}(x), y;)) \quad \mathbf{i} \neq \mathbf{j}
\end{aligned}$$

Chaque règle s'ordonne en posant $\texttt{max} \prec_{\mathcal{F}} \texttt{lcs}$. Le programme `max` calcule le maximum de deux entiers s'ils sont inférieurs à la taille de l'argument de récurrence, *i.e.* $[\![\texttt{max}]\!](u, n, m) = \max(n, m)$ si $n, m \leq |u|$. Nous reviendrons dans la section 9.2 sur cette bizarrerie. La complexité de cet algorithme récursif est exponentielle. Cependant, si nous utilisons la programmation dynamique, il est alors bien connu que ce problème se résout en temps polynomial, plus exactement en temps $O(n^2)$ où n est la longueur du plus grand mot d'entrée. Cet exemple est traité dans la plupart des livres d'algorithmique, comme dans le livre de Cormen, Leiserson et Rivest [CLR90].

9.1.4 Expressivité

Tout d'abord, nous allons restreindre le calcul aux mots, aux listes. Pour cela, nous dirons qu'une sorte $\mathbf{s}$ est *simple* si chaque constructeur d'arité de type $\mathbf{s}_1, \ldots, \mathbf{s}_n \to \mathbf{s}$, la sorte $\mathbf{s}$ apparaît au plus une fois dans $(\mathbf{s}_1, \ldots, \mathbf{s}_n)$.

Définition 9.8. La classe de programmes $\mathrm{LPRO}_{\mathrm{LIN}}$ est l'ensemble des programmes, définis sur des sortes simples, et ordonnés par LPRO tel que pour chaque règle $l \to r$ qui définit un symbole de fonction `f` qui a un statut lexicographique, *i.e.* $\tau(\texttt{f}) = \ell$, alors `f` apparaît au plus une fois dans r.

Théorème 9.9. *La classe des fonctions* $\mathcal{F}(\mathrm{LPRO}_{\mathrm{LIN}})$ *est exactement la classe* PTIME *des fonctions calculables en temps polynomial.*

Le résultat est obtenu en construisant un interpréteur efficace dont la description est reporté à la section 9.4.

La condition sur les symboles de fonction de statut lexicographique est nécessaire. Si nous considérons les programmes définis suivant le schéma d'une récurrence avec substitutions de paramètres et qui sont ordonnés lexicographiquement, alors ces programmes définissent toutes les fonctions calculables en espace polynomial, suivant le rapport de Cichon et Marion [CM00].

9.1.5 Une analyse de la complexité implicite

Arrêtons-nous un instant pour discuter du résultat de ce chapitre. Notre intention n'est pas d'obtenir une modélisation supplémentaire des fonctions de PTIME, mais plutôt une analyse des algorithmes. Dans sa thèse [Cas97], Caseiro avait abordé la même problématique.

Reprenons l'exemple du programme `lcs`. Si un calcul de `lcs` s'effectue suivant la sémantique usuelle de réécriture voir le chapitre 1, alors l'algorithme `lcs` est exponentiel car la longueur des dérivations l'est. Or, nous savons que la *fonction* $[\![\texttt{lcs}]\!]$ est calculable en temps $O(n^2)$. Le résultat affirme que la

fonction $[\![\texttt{lcs}]\!]$ est calculable en temps polynomial. Cette étude repose sur deux aspects.

1. Une analyse des programmes par leur preuve de terminaison qui est obtenue par un couple d'ordres $(\prec_1, \prec_0)$. Ces ordres sont des restrictions d'ordres bien connus qui capturent des schémas d'algorithmes. La combinaison avec les concepts tirés de la récurrence ramifiée a permis de cerner la complexité intrinsèque de l'algorithme.
2. Un interpréteur qui évalue efficacement les programmes de LPRO. Cet interpréteur met en mémoire les résultats intermédiaires obtenus. Nous reportons sa description à la section 9.4.

D'un point vue méthodologique, nous voyons le programme source comme une spécification algébrique, décrivant un algorithme avec un haut-degré d'abstraction. Nous proposons une analyse de la complexité de la fonction calculée par cette spécification. Cette analyse peut amener à transformer le programme source afin de compiler un programme plus efficace.

Le travail exposé est encore à un stade préliminaire. La combinaison de l'ordre multi-ensemble et de l'ordre lexicographique permet de capturer certains schémas d'algorithmes. L'ajout d'autres ordres qui étendent les comparaisons sur les suites d'arguments devrait permettre d'augmenter la classe de schémas d'algorithmes.

Une autre orientation est d'augmenter la classe des algorithmes avec des constructions d'ordre supérieur, ce qui a été explorée par Bellantoni, Niggl, Schwichtenberg [BNS00], et Hofmann [Hof97, Hof99c, Hof99b],

Enfin, Leivant [Lei99a] a tracé une autre voie à mi-chemin entre l'analyse prédicative et l'approche plus structurelle de Jones [Jon99, Jon00].

9.2 Fonctions non-croissantes

Toujours dans l'exemple de `lcs`, le programme `max` comporte 3 arguments et le premier argument semble de trop. Sa présence est rendue nécessaire par les valences imposées à `lcs`. En fait, il faut comprendre cet argument comme une "obligation de preuve", car il indique que sur le domaine de calcul de `lcs`, nous avons toujours $[\![\texttt{lcs}(x, y)]\!] \leq |x|$.

Dans l'équation du tri par insertion, l'analyse prédicative interdit à l'argument critique $\texttt{sort}(l)$ d'être le paramètre d'une récursion. Or, `insert` est défini par récurrence sur son deuxième argument. Il s'ensuit que `sort` n'est pas ramifié.

La raison de cette difficulté se trouve dans la dernière équation de `lcs` que nous ré-écrivons ci-dessous.

$$\texttt{lcs}(\mathbf{i}(x), \mathbf{j}(y)) \to \texttt{max}(\texttt{lcs}(x, \mathbf{j}(y)), \texttt{lcs}(\mathbf{i}(x), y))$$

L'analyse prédicative interdit à l'argument critique $\texttt{lcs}(x, \mathbf{j}(y))$ ou $\texttt{lcs}(\mathbf{i}(x), y)$ d'être le paramètre d'une récursion. Or, la fonction maximun est définie par récurrence sur au moins un de ses arguments. D'un point de vue algorithmique, cette situation n'est pas acceptable car l'opération qui consiste à prendre le maximun de deux arguments est sûre. En effet, la taille du résultat est inférieure ou égale à la taille des arguments.

Cette critique de l'analyse prédicative a été menée par Hofmann [Hof99a] et repris par Aehlig et Schwichtenberg dans [AS00] dans le cadre de la récursion primitive simple. Pour expliquer la réponse apportée, nous allons prendre pour exemple l'équation critique d'une définition par récurrence sur les entiers.

$$\mathtt{f}(\mathbf{suc}(t), \vec{y}) \to \mathtt{h}(\vec{y}, \mathtt{f}(t, \vec{y}))$$

A chaque appel récursif, `f` libère une ressource en détruisant un symbole **suc**. Le programme `h` n'a le droit de consommer que la quantité de ressource libérée, pour construire un nouvel entier. Ainsi, la quantité de ressource est globalement invariante. C'est pour cela que Hofmann a qualifié ces fonctions de non-croissantes (*non-size-increasing*). En s'inspirant de la logique linéaire, il a développé un système de types pour les fonctions non-croissantes. Ce système de types permet de contrôler l'allocation de la mémoire. Dans l'article [Hof00], Hofmann développe un compilateur pour les programmes qui calculent des fonctions non-croissantes. Le programme compilé ne réclame pas de mémoire supplémentaire.

Il semble possible d'adapter le système de types de Hofmann pour analyser les programmes comme `lcs`, *i.e.* un programme de LPRO, et de les compiler en combinant la mise en mémoire des appels récursifs et la non-allocaction de mémoire supplémentaire.

Une autre manière de contourner cette difficulté est d'utiliser un argument sémantique que nous allons exposer maintenant.

9.3 Quasi-interprétation polynomiale

Nous reportons le résultat dû à Marion et Moyen [MM00b] auquel nous ajoutons la possibilité d'utiliser le statut lexicographique.

Définition 9.10. Une quasi-interprétation $(\!|\ |\!)$ associe à chaque symbole $f \in \mathcal{C} \cup \mathcal{F}$ d'arité n, une fonction $(\!|f|\!) : \mathbb{N}^n \to \mathbb{N}$ qui vérifie

1. $(\!|f|\!)$ est croissante suivant chaque argument, *i.e.* pour tout $1 \leq i \leq n$

$$X_i \leq Y_i \text{ implique } (\!|f|\!)(\ldots, X_i, \ldots) \leq (\!|f|\!)(\ldots, Y_i, \ldots)$$

2. $(\!|f|\!)(\ldots, X, \ldots) \geq X$.

Un terme clos $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{C}, \mathcal{F})$ est quasi-interprété par

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

Définition 9.11. Un programme $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ admet une quasi-interprétation $(\!|\ |\!)$ si pour toute équation $l \to r \in \mathcal{E}$, et pour toute substitution close σ, on a $(\!|r\sigma|\!) \leq (\!|l\sigma|\!)$.

Bien entendu, et à la différence des interprétations définies au chapitre 8, une quasi-interprétation ne prouve pas la terminaison d'un programme.

Définition 9.12. Un programme $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ admet une quasi-interprétation polynomiale $(\!|\ |\!)$ si

1. pour tout $\mathtt{f} \in \mathcal{F}$, la fonction $(\!|\mathtt{f}|\!)$ est bornée par un polynôme,

2. pour tout $\mathbf{c} \in \mathcal{C}$, $(\!|\mathbf{c}|\!)(X_1, \ldots, X_n) = \sum_{1 \leq i \leq n} X_i + \gamma$, où $\gamma > 0$.

Remarque 9.13. Nous considérons une extension de la classe $\Pi(0)$ développée au chapitre 8. Ici, la quasi-interprétation ne mesure pas la taille d'un terme obtenu dans une dérivation. Dans l'exemple de `lcs` ci-dessous, nous savons que la taille des termes d'une dérivation est exponentielle, bien que `lcs` ait une quasi-interprétation polynomiale.

Définition 9.14.

- Un programme est ordonné par PRO s'il est ordonné par RPO avec un statut limité aux extensions produit et lexicographique.
- La classe de programme $\mathrm{PRO}_{\mathrm{LIN}}$ est l'ensemble des programmes ordonnés par PRO tel que pour chaque règle $l \to r$ qui définit un symbole de fonction `f` qui a un statut lexicographique, *i.e.* $\tau(\mathtt{f}) = \ell$, alors `f` apparaît au plus une fois dans r.
- La classe de programmes $\mathrm{PRO}^{\mathrm{POL}}_{\mathrm{LIN}}$ est constituée des programmes de $\mathrm{PRO}_{\mathrm{LIN}}$ qui admettent une quasi-interprétation polynomiale.

Théorème 9.15. *La classe* $\mathcal{F}(\mathrm{PRO}^{\mathrm{POL}}_{\mathrm{LIN}})$ *est exactement la classe* PTIME *des fonctions calculables en temps polynomial.*

Exemple 9.16.

1. Les programmes `inf`, `Linf` et `renverser` appartiennent à $\mathrm{PRO}^{\mathrm{POL}}_{\mathrm{LIN}}$.
2. Nous pouvons programmer `lcs` sans incorporer un argument supplémentaire comme "obligation de preuve".

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

   Le programme est ordonné par PRO et les symboles de fonctions ont tous un statut produit. Encore une fois, nous livrons au lecteur courageux le soin de vérifier que le programme `lcs` est quasi-interprété comme suit. Tout comme dans le cas de la terminaison par interprétation polynomiale, il n'y a pas de méthode pour trouver la bonne quasi-interprétation, il faut alors procéder par heuristiques.

$$
\begin{aligned}
(\!|\mathbf{0}|\!) = (\!|\boldsymbol{\epsilon}|\!) &= 1 \\
(\!|\mathbf{suc}|\!)(X) = (\!|\mathbf{0}|\!)(X) = (\!|\mathbf{1}|\!)(X) &= X + 1 \\
(\!|\mathtt{max}|\!)(X, Y) = (\!|\mathtt{lcs}|\!)(X, Y) &= \max(X, Y)
\end{aligned}
$$

3. Le tri par insertion appartient à $\mathrm{PRO}^{\mathrm{POL}}_{\mathrm{LIN}}$.

$\mathtt{if\_then\_else} : \mathbf{bool}, \alpha, \alpha \to \alpha$

$$
\begin{aligned}
\mathtt{if}\ \mathbf{tt}\ \mathtt{then}\ x\ \mathtt{else}\ y &\to x \\
\mathtt{if}\ \mathbf{ff}\ \mathtt{then}\ x\ \mathtt{else}\ y &\to y
\end{aligned}
$$

$<: \mathbf{nat}, \mathbf{nat} \to \mathbf{bool}$

$$\begin{aligned} \mathbf{0} < \mathbf{suc}(y) &\to \mathbf{tt} \\ x < \mathbf{0} &\to \mathbf{ff} \\ \mathbf{suc}(x) < \mathbf{suc}(y) &\to x < y \end{aligned}$$

$\mathtt{insert} : \mathbf{nat}, \mathbf{list}(\mathbf{nat}) \to \mathbf{list}(\mathbf{nat})$

$$\begin{aligned} \mathtt{insert}(a, \boldsymbol{\epsilon}) &\to \mathbf{cons}(a, \boldsymbol{\epsilon}) \\ \mathtt{insert}(a, \mathbf{cons}(b, l)) &\to \mathtt{if}\ a < b\ \mathtt{then}\ \mathbf{cons}(a, \mathbf{cons}(b, l)) \\ &\qquad \mathtt{else}\ \mathbf{cons}(b, \mathtt{insert}(a, l)) \end{aligned}$$

$\mathtt{sort} : \mathbf{list}(\mathbf{nat}) \to \mathbf{list}(\mathbf{nat})$

$$\begin{aligned} \mathtt{sort}(\boldsymbol{\epsilon}) &\to \boldsymbol{\epsilon} \\ \mathtt{sort}(\mathbf{cons}(a, l)) &\to \mathtt{insert}(a, \mathtt{sort}(l)) \end{aligned}$$

Le programme admet la quasi-interprétation suivante.

$$\begin{gathered} (\!|\mathbf{tt}|\!) = (\!|\mathbf{ff}|\!) = (\!|\boldsymbol{\epsilon}|\!) = (\!|\mathbf{0}|\!) = 1 \\ (\!|\mathbf{suc}|\!)(X) = (\!|\mathbf{cons}|\!)(X) = X + 1 \\ (\!|\mathtt{if\_then\_else}|\!)(X, Y, Z) = \max(X, Y, Z) \\ (\!|<|\!)(X, Y) = \max(X, Y) \\ (\!|\mathtt{insert}|\!)(X, Y) = X + Y + 1 \\ (\!|\mathtt{sort}|\!)(X) = X \end{gathered}$$

9.4 Un interpréteur efficace

Les programmes de $\mathrm{PRO}_{\mathrm{LIN}}$ et de $\mathrm{PRO}^{\mathrm{POL}}_{\mathrm{LIN}}$ sont exécutés par l'interpréteur décrit dans la figure 9.3. Cet interpréteur évalue un programme par appel par valeur en mémorisant les résultats intermédiaires dans un cache C.

Appelons configuration un couple $(\mathtt{g}(\mathbf{d}_1, \ldots, \mathbf{d}_n), \mathbf{e})$ où $(\mathbf{d}_1, \ldots, \mathbf{d}_n)$ et $\mathbf{e}$ sont dans $\mathcal{T}(\mathcal{C})$ et $[\![\mathtt{g}]\!](\mathbf{d}_1, \ldots, \mathbf{d}_n) = \mathbf{e}$. Un cache est une liste de configurations. Quand l'interpréteur exécute $\mathtt{f}(t_1, \ldots, t_n)$, il calcule récursivement les valeurs de chaque argument t_i. Disons que $[\![t_i]\!] = \mathbf{d}_i$. Ensuite, il vérifie s'il y a une configuration $(\mathtt{f}(\mathbf{d}_1, \ldots, \mathbf{d}_n), \mathbf{e})$ qui est dans le cache C. Si c'est le cas, alors le résultat est $\mathbf{e}$, et le calcul est court-circuité. Sinon, il faut continuer en appliquant une équation du programme. Une fois, le résultat de $\mathtt{f}(\mathbf{d}_1, \ldots, \mathbf{d}_n)$ connu, il est inséré dans le cache.

Proposition 9.17. *Soit $f = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \Gamma \rangle$ un programme. Pour toute entrée $\vec{\mathbf{a}} \in \mathcal{T}(\mathcal{C})$, nous avons*

1. *$\mathcal{E}, \emptyset \vdash \langle \emptyset, f(\vec{\mathbf{a}}) \rangle \to \langle C, \mathbf{e} \rangle$ si, et seulement si, $[\![f]\!](\vec{\mathbf{a}}) = \mathbf{e}$*
2. *le calcul de $\mathcal{E}, \emptyset \vdash \langle \emptyset, f(\vec{\mathbf{a}}) \rangle \to \langle C, \mathbf{e} \rangle$ s'effectue en temps polynomial dans la taille des entrées $\vec{\mathbf{a}}$.*

$$\frac{\sigma(x) = \mathbf{e}}{\mathcal{E}, \sigma \vdash \langle C, x\rangle \to \langle C, \mathbf{e}\rangle}$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, \mathbf{d}_i\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathbf{c}(t_1, \ldots, t_n)\rangle \to \langle C_n, \mathbf{c}(\mathbf{d}_1, \ldots, \mathbf{d}_n)\rangle}$$

$$\frac{\mathtt{f} \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, \mathbf{d}_i\rangle \quad (\mathtt{f}(\mathbf{d}_1, \ldots, \mathbf{d}_n), \mathbf{e}) \in C_n}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \ldots, t_n)\rangle \to \langle C_n, \mathbf{e}\rangle}$$

$$\frac{\mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, \mathbf{d}_i\rangle \quad \mathtt{f}(\vec{p}) \to r \in \mathcal{E} \quad p_i\sigma' = \mathbf{d}_i \quad \mathcal{E}, \sigma' \vdash \langle C_n, r\rangle \to \langle C, \mathbf{e}\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \ldots, t_n)\rangle \to \langle C \cup (\mathtt{f}(\mathbf{d}_1, \ldots, \mathbf{d}_n), \mathbf{e}), \mathbf{e}\rangle}$$

FIG. 9.3 – Interpréteur par appel par valeur avec un cache C, étant donné un programme $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \Gamma\rangle$, et une assignation σ des variables

.

Remarque 9.18. Cette technique de mise mémoire des résultats partiels s'apparente à la programmation dynamique. Ce travail s'inspire directement des travaux de Jones [Jon99, Jon00]. L'idée de mise en mémoire des résultats partiels remonte aux travaux [Coo71] de Cook sur les k-2PDA. Un k-2PDA est un automate à pile avec k têtes de lecture qui peuvent se déplacer aussi bien vers la gauche que vers la droite. Cook a démontré que les langages reconnus par k-2PDA sont les langages décidés en temps polynomial. La simulation des k-2PDA par une machine de Turing nécessite le stockage de résultats partiels du calcul d'un k-2PDA. Plus récemment, les travaux de Liu et Stoller [LS99] utilise la programmation dynamique dans la compilation.

9.4.1 Minimiser le cache

L'interpréteur mémorise toutes les valeurs intermédiaires nécessaires au calcul. Pouvons-nous diminuer le nombre de résultats mémorisés ?

Disons qu'un symbole de fonction $\mathtt{f}$ est affine si pour chaque équation $\mathtt{f}(p_1, \ldots, p_n) \to r$, $\mathtt{f}$ apparaît au plus une fois dans r. Par exemple, les symboles de fonctions qui ont un statut lexicographique sont affines. La première amélioration consiste à ne pas mettre dans le cache les résultats provenant du calcul d'un symbole de fonction affine.

Mais, nous pouvons faire beaucoup mieux. Pour cela, interessons-nous à l'évaluation de $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ où $\mathbf{a}_i \in \mathcal{T}(\mathcal{C})$ et le statut de $\mathtt{f}$ est produit. Maintenant, disons qu'un *appel* est un terme $\mathtt{f}(\mathbf{d}_1, \ldots, \mathbf{d}_n)$ qui est calculé par l'interpréteur pour obtenir $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$.

La *couverture* de $\mathtt{f}$ est l'ensemble des couples $(\mathtt{f}(p_1, \ldots, p_n), \mathtt{f}(s_1, \ldots, s_n))$ tels que $\mathtt{f}(p_1, \ldots, p_n) \to r$ est une équation et $\mathtt{f}(s_1, \ldots, s_n)$ est un sous-terme de r. Il est important d'observer que chaque s_i est un motif, *i.e.* $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, car nous avons $\mathtt{f}(s_1, \ldots, s_n) \prec_0 \mathtt{f}(p_1, \ldots, p_n)$.

Le *graphe de couverture* de $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ est défini inductivement comme

suit. $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ est la racine du graphe. Si u est un nœud et s'il existe un couple $(\mathtt{f}(p_1, \ldots, p_n), \mathtt{f}(s_1, \ldots, s_n))$ de la couverture de $\mathtt{f}$, ainsi qu'une substitution σ, telles que $u = \mathtt{f}(p_1, \ldots, p_n)\sigma$, alors $(u, \mathtt{f}(s_1, \ldots, s_n)\sigma)$ est un arc. Dans ce cas, nous dirons que u *couvre* $\mathtt{f}(s_1, \ldots, s_n)\sigma$. Le graphe de couverture de $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ est l'ensemble des appels nécessaires au calcul de la racine $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$. L'évaluation d'un nœud nécessite la connaissance des résultats des calculs des nœuds qui le couvrent.

Un *filet* F est un ensemble de nœud du graphe de couverture tel que chaque chaîne qui part de la racine rencontre un nœud du filet. Supposons que nous connaissions les valeurs des appels d'un filet F. Dès lors, les résultats qui sont couverts par un nœud du filet, sont inutiles. Donc, pour calculer $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$, il suffit de garder les résultats d'un filet minimal. Un filet F est minimal si pour tout $x \in F$, $F \setminus \{x\}$ n'est plus un filet. La taille d'un filet minimal est inférieure ou égale au maximum de la taille des antichaînes du graphe de couverture de $\mathtt{f}(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ (ou par le plus petit nombre de chaînes pour couvrir le graphe, suivant le théorème de Dilworth [Dil50]). Il n'est pas difficile de maintenir un cache minimal durant l'éxécution de l'interpréteur de la figure 9.3.

Chapitre 10

Conclusion et perspectives

10.1 Preuve, calcul et complexité

La principale contribution de cette habilitation est de proposer des méthodes, inspirées de la théorie de la démonstration pour analyser la complexité des algorithmes, que nous avons baptisée *complexité implicite des calculs* (CIC). Un des succès de l'analyse de la CIC est d'offrir la possibilité de transformer un programme en un programme plus efficace. Certes, ni l'analyse de la complexité des programmes, ni les méthodes de transformation de programmes ne sont nouvelles. Les autres travaux sur l'analyse de la complexité des programmes, comme ceux de Flajolet, Salvy et Zimmermann [FSZ90] ou de Le Métayer [Mét88], déterminent la complexité, généralement par des moyens sophistiqués de calcul symbolique. Plus récemment, Benzinger [Ben99] a développé un analyseur de la complexité d'un programme Nuprl. Crary et Weirich [CW00] ont décrit un système de types dépendants pour contrôler la complexité d'un programme. Ces travaux s'inscrivent dans la certification de l'efficacité d'un programme. Cependant, il n'est pas possible de transformer un programme lorsqu'il s'avère trop gourmand en ressources de calcul, et, dans ce cas, il faut redévelopper le programme.

Les travaux sur les transformations de programmes forment une formidable collection de recettes pour améliorer les programmes, voir le livre de Jones, Gomard et Sestoft [JGS93]. Mais, la certification des ressources nécessaires aux calculs est ignorée.

En revanche, notre approche se fonde sur une analyse d'une démonstration de terminaison d'un programme, par le biais de la théorie des types ou des ordres de terminaison. Les informations de l'analyse peuvent nous conduire à améliorer un programme, nous avons obtenu un gain de temps exponentiel sur certains exemples.

10.2 Vers une programmation intensionnelle

L'objectif de la CIC est de pouvoir certifier la qualité d'un programme. D'un point de vue méthodologique, l'analyse de la CIC est un moyen de compiler un algorithme écrit avec un haut degré d'abstraction, comme une spécification algébrique, en un programme certifié et efficace. C'est un premier pas

vers ce que nous pourrions appeler la programmation intensionnelle.

Le travail devant nous est encore immense. La validation de l'analyse de la CIC comme outil réel, nécessite la prise en compte de plus de schémas d'algorithme (travail en cours) et l'intégration de plus d'outils de transformations de programmes (évaluation partielle, déforestation, gestion de la mémoire, parallélisation). Les domaines d'applications sont variés, citons trois exemples :

1. La programmation fonctionnelle au premier ordre, comme dans le langage de programmation Elan.
2. Les programmes certifiés qui sont obtenus semi-automatiquement à partir d'une démonstration de sa spécification comme dans les systèmes Alf, Coq ou Nuprl.
3. La certification des ressources de calcul pour les programmes mobiles et embarqués.

Très clairement, pour achever nos objectifs, il est nécessaire de disposer d'une théorie des algorithmes aux fondements mathématiques solides, et nous n'en sommes qu'au début.

Index

Bibliographie

[Ack28] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. annalen*, 99 :118–133, 1928.

[Alf] *Alf.* `http ://www.cs.chalmers.se/Cs/Research/Logic/alf/guide.html`.

[AS00] K. Aehlig and H. Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings of the Fifteenth IEEE Symposium on Logic in Computer Science (LICS '00)*, pages 84 – 91, 2000.

[Asp98] A. Asperti. Light affine logic. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science(LICS'98)*, pages 300–308, 1998.

[Bar80] H. Barendregt. *The Lambda-Calculus : Its Syntax and Semantics*. North-Holland, Amsterdam, 1980.

[BB85] C. Bohm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39 :135–154, 1985.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2 :97–110, 1992.

[BCMT99] G. Bonfante, A. Cichon, J-Y Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *Computer Science Logic, 12th International Workshop, CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384, 1999.

[BCMT00] G. Bonfante, A. Cichon, J-Y Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11, 2000.

[Bel94] S. Bellantoni. Predicative recursion and the polytime hierarchy. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, *Perspectives in Computer Science*. Birkhäuser, 1994.

[Ben99] R. Benzinger. *Automated complexity analysis of NUPRL extracts*. PhD thesis, Cornell University, 1999.

[BH00] S. Bellantoni and M. Hofmann. A new "feasible" arithmetic. *Journal of symbolic logic*, 2000. A paraître.

[Blo94] S. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational complexity*, 4(2) :175–205, 1994.

[BM96] P. Bradford and J-Y Marion. A parallel decision procedure for ALL. Proceedings of the Workshop on Proof Search in Type-Theoretic Languages affiliated to CADE 13, 1996.

[BN99] S. Bellantoni and K-H Niggl. Ranking primitive recursions : The low Grzegorczyk classes revisited. *SIAM Journal on Computing*, 29(2) :401–415, 1999.

[BNS00] S. Bellantoni, K-H Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3) :17–30, 2000.

[Bon00] G. Bonfante. *Constructions d'ordres, analyse de la complexité*. PhD thesis, Institut National Polytechnique de Lorraine, 2000.

[Bus86] S. Buss. *Bounded arithmetic*. Bibliopolis, 1986.

[BW96] A. Beckmann and A. Weiermann. Characterizing the elementary recursive functions by a fragment of gödel´s t. *Archive for Mathematical Logic*, 1996. A paraître.

[Ca86] R. Constable and al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986. `http ://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

[Cas97] V.-H. Caseiro. *Equations for defining Poly-Time*. PhD thesis, University of Oslo, 1997.

[CB98] E.A. Cichon and E. Tahhan Bittar. Ordinal recursive bounds for higman's theorem. *Theoretical Computer Science*, 201(1-2) :63–84, 1998.

[CF93] T. Coquand and D. Fridlender. A proof of Higman's lemma by structural induction. http ://www.cs.chalmers.se/∼coquand/intuitionism.html, 1993.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76 :95–120, 1988.

[CL87] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of computer Programming*, pages 131–159, 1987.

[CL92] E.A Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *CADE'11*, number 607 in Lecture Notes in Artificial Intelligence, pages 139–147, 1992.

[Clo95] P. Clote. Computational models and function algebras. In D. Leivant, editor, *LCC'94*, volume 960 of *Lecture Notes in Computer Science*, pages 98–130, 1995.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CM00] A. Cichon and J-Y Marion. The light lexicographic path ordering. Technical report, Loria, 2000. Workshop Rule.

[Cob62] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.

[Col98] L. Colson. Functions versus algorithms. *EATCS Bulletin*, 65, 1998. The logic in computer science column.

[Col99] L. Colson. Les systèmes fonctionnels : problèmes pratiques et théoriques, 1999. Habilitation.

[Con95] R. Constable. Expressing computationnal complexity in constructive type theory. In D. Leivant, editor, *LCC'94*, volume 960 of *Lecture Notes in Computer Science*, pages 131–144, 1995.

[Coo71] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1) :4–18, January 1971.

[Coq] *COQ*. `http ://pauillac.inria.fr/coq/coq-eng.html`.

[Cou90] B. Courcelle. *Handbook of theoretical computer science*, chapter Recursive applicative program schemes, pages 461–492. Elsevier, 1990.

[ÇOW] N. Çağman, G. Ostrin, and S. Wainer. Proof theoretic complexity of low subrecursive classes.

[CPC00a] S. Caporaso, G. Pani, and E. Covino. Predicative recursion, constructive diagonalization and the elementary functions. *Information and Computation*, 2000. A paraître.

[CPC00b] E. Covino, G. Pani, and S. Caporaso. Extending the implicit computational complexity approach to the sub-elementary time-space classes. In *CIAC*, volume 1767 of *Lecture Notes in Computer Science*, pages 239–252, 2000.

[CW83] A. Cichon and S. Wainer. The slow-growing and the Grzegorczyk hierarchies. *Journal of symbolic logic*, 48(2) :399–408, 1983.

[CW00] K. Crary and S. Weirich. Ressource bound certification. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL*, pages 184 – 198, 2000.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3) :279–301, 1982.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, pages 69–115, 1987.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math.*, (51) :161–166, 1950.

[DJ90] N. Dershowitz and J-P Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. Elsevier Science Publishers B. V. (NorthHolland), 1990.

[Dow99] G. Dowek. La part du calcul, 1999. Habilitation.

[Fag74] R. Fagin. Generalized first order spectra and polynomial time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS, 1974.

[FLO83] S. Fortune, D. Leivant, and M. O'Donnell. The expressiveness of simple and second order type structures. *Journal of the ACM*, 30 :151–185, 1983.

[FSZ90] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. Technical Report 1233, INRIA, 1990.

[Gal91] J. Gallier. What's so special about Kruskal theorem and the ordianl Γ_0. *Annals of Pure and Applied Logic*, 53 :199–260, 1991.

[Gen35] G. Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North Holland, 1935.

[GG95] E. Grädel and Y. Gurevich. Tailoring recursion for complexity. *Journal of Symbolic Logic*, 60(3) :952–969, Sept. 1995.

[Gie95] J. Giesl. Generating polynomial orderings for termination proofs. In *RTA*, number 914 in Lecture Notes in Computer Science, pages 427–431, 1995.

[Gir72] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre superieur, 1972. Thèse de Doctorat d'Etat.

[Gla67] M.D. Gladstone. A reduction of the recursion scheme. *Journal of symbolic logic*, 32(4) :505–508, 1967.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge university press, 1989.

[GM95a] D. Galmiche and J-Y Marion. Dealing with additives in MALL proof search (short abstract). Proceedings of the Workshop on Non-Standart Logic, June 1995.

[GM95b] D. Galmiche and J-Y Marion. Semantic methods for ALL - a first approach. Proceedings of the Tableaux Workshop (short papers), May 1995.

[GM00] S. Grigorieff and J-Y Marion. Kolmogorov complexity and non-determinism. *Theoretical Computer Science*, 2000. A paraître.

[Göd58] Kurt Gödel. Über eine bisher noch nicht benüte Erweiterung des finiten Standpunktes. *Dialectica*, 12 :280–287, 1958. Republished with English translation and explanatory notes by A. S. Troelstra in *Kurt Gödel : Collected Works*, Vol. II. S. Feferman, ed. Oxford University Press, 1990.

[Goe92a] A. Goerdt. Characterizing complexity classes by general recursive definitions in higher types. *Information and Computation*, 101(2) :202–218, 1992.

[Goe92b] A. Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100(1) :45–66, 1992.

[Grz53] A. Grzegorczyk. Some classes of recursive functions. In *Rozprawy Mate. IV*. Warsaw, 1953.

[Gur83] Y. Gurevich. Algebras of feasible functions. In *Twenty Fourth Symposium on Foundations of Computer Science*, pages 210–214. IEEE Computer Society Press, 1983.

[Gur88] Y. Gurevich. Kolmogorov machines and related issues. *EATCS Bulletin*, 33 :71–82, 1988.

[Gur99] Y. Gurevich. The sequential ASM thesis. *EATCS Bulletin*, 67 :93–124, feb 1999.

[Hei61] W. Heinermann. *Untersuchungen über die rekursionszahlen rekursiver funktionen*. PhD thesis, Universität Münster, 1961.

[Hen60] L. Henkin. On mathematical induction. *American Mathematical Monthly*, pages 323–338, 1960.

[Hig52] G. Higman. Ordering by divisibility in abstract algebra. *Proc. London Mathematical Society*, 3(2), 1952.

[HL88] D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *RTA*, number 355 in Lecture Notes in Computer Science, 1988.

[Hof92] D. Hofbauer. Termination proofs with multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1) :129–140, 1992.

[Hof97] M. Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *CSL*, pages 275–294, 1997.

[Hof99a] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.

[Hof99b] M. Hofmann. Semantics of linear/modal lambda calculu. *Journal of Functional Programming*, 9(3) :247–277, 1999.

[Hof99c] M. Hofmann. Type systems for polynomial-time computation, 1999. Habilitation.

[Hof00] M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.

[HS86] R. Hindley and J. Seldin. *Introduction to combinators and λ-calculus*. Cambridge university press, 1986.

[Hue80] G. Huet. Confluent reductions : Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4) :797–821, 1980.

[Imm99] N. Immerman. *Descriptive Complexity*. Springer, 1999.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993. With chapters by L.O. Andersen and T. Mogensen. accessible `http ://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[Jon93] N. Jones. Constant time factors do matter. In *Symposium on Theory of Computing Proceedings*, pages 602–611. ACM, 1993.

[Jon97] N. Jones. *Computability and complexity, from a programming perspective*. MIT press, 1997.

[Jon99] N. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 228 :151–174, 1999.

[Jon00] N. Jones. The expressive power of higher order types or, life without cons. A paraître, 2000.

[JYG92] P. Scott J.-Y. Girard, A. Scedrov. A modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1) :1–66, 1992.

[KK] C. Kirchner and H. Kirchner. Rewriting solving proving. accessible `http ://www.loria.fr/∼ckirchne`.

[KL80] S. Kamin and J-J Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerity of Illinois, Urbana, 1980. Unpublished note.

[Kle52] S.C. Kleene. *Introduction to metamathematics*. North-Holland, 1952.

[KP87] J.L. Krivine and M. Parigot. Programming with proofs. In *SCT'87*, pages 145–159, 1987.

[Kre88] M. Krentel. The complexity of optimization problems. *Journal of computer and system sciences*, 36 :490–519, 1988.

[Kri90] J.-L. Krivine. *Lambda-calcul*. Masson, 1990.

[Kru60] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95 :210–225, 1960.

[KU58] A.N. Kolmogorov and V. Uspensky. To the definition of an algorithm. *Uspekhi Mat. Naut*, 13(4), 1958. English translation in AMS translation vol. 21 (1963), 217-245.

[Lan79] D.S. Lankford. On proving term rewriting systems are noetherien. Technical Report MTP-3, Louisiana Technical University, 1979.

[Lau88] C. Lautemann. A note on polynomial interpretation. *EATCS Bulletin*, 4 :129–131, Oct 1988.

[Lei83] D. Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Twenty Fourth Symposium on Foundations of Computer Science*, pages 460–469, 1983.

[Lei90] D. Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 279–327. Academic Press, London, 1990.

[Lei91] D. Leivant. A foundational delineation of computational feasiblity. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91)*, 1991.

[Lei94a] D. Leivant. A foundational delineation of poly-time. *Information and Computation*, 110(2) :391–420, 1994.

[Lei94b] D. Leivant. Intrinsic theories and computational complexity. In *Logical and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 1994.

[Lei94c] D. Leivant. Predicative recurrence and computational complexity I : Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.

[Lei98] D. Leivant. A characterization of nc by tree recurrence. In *39th Annual Symposium on Foundations of Computer Science, FOCS'98*, pages 716–724, 1998.

[Lei99a] D. Leivant. Applicative control and computational complexity. In *Computer Science Logic, 13th International Workshop, CSL '99*, volume 1683 of *Lecture Notes in Computer Science*, pages 82–95, 1999.

[Lei99b] D. Leivant. Ramified recurrence and computational complexity III : Higher type recurrence and elementary complexity. *Annals of Pure and Applied Logic*, 96(1-3) :209–229, 1999.

[Lei00] D. Leivant. Intrinsic reasoning about functional programs I : first order theories. *Annals of Pure and Applied Logic*, 2000. A paraître.

[Les92] P. Lescanne. Termination of rewrite systems by elementary interpretations. In H. Kirchner and G. Levi, editors, *3rd International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 1992.

[Lev73] L. Levin. Universal search problems. *Problems of information transmission*, 9(3) :265–266, 1973.

[LM93a] D. Leivant and J-Y Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19(1,2) :167,184, September 1993.

[LM93b] D. Leivant and J-Y Marion. A λ-calculus characterizations of poly-time. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 1993.

[LM95] D. Leivant and J-Y Marion. Ramified recurrence and computational complexity II : substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500, Kazimierz,Poland, 1995. Springer.

[LM97] D. Leivant and J-Y Marion. Predicative functional recurrence and poly-space. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97, Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 369–380. Springer, Apr 1997.

[LM00a] D. Leivant and J-Y Marion. A characterization of alternating log time by ramified recurrence. *Theoretical Computer Science*, 236(1-2) :192–208, Apr 2000.

[LM00b] D. Leivant and J-Y Marion. Predicative recurrence and computational complexity IV : Predicative functionals and poly-space. *Information and Computation*, 2000. A paraître.

[LS99] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *Proceedings of the 8th European Symposium on Programming*, pages 288–305, Amsterdam, The Netherlands, March 1999. Springer.

[LV97] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Application*. Springer, 1997. (Second edition).

[Mar91] J-Y Marion. *Extension du système T de Gödel dans les domaines finis, Etude de la complexité*. PhD thesis, Université Paris 7, December 1991. 140 pages.

[Mar96] J-Y Marion. Additive linear logic with analytic cut. *IGPL*, 4(3) :48–51, May 1996. Third Worshop on Logic, Language, Information and Computation. R. de Queiroz. `http ://www.mpi-sb.mpg.de/igpl/Bulletin/V4-3/`.

[Mar97] J-Y Marion. Case study : Additive linear logic and lattices. In S. Adian A. Nerode, editor, *4th International Symposium Logical, Foundations of Computer Science*, volume 1234 of *Lecture Notes in Computer Science*, pages 237–247. Springer, July 1997.

[Mar99] J-Y Marion. From multiple sequent for additive linear logic to decision procedures for free lattices. *Theoretical Computer Science*, 224(1-2) :157–172, 1999.

[Mar00] J-Y Marion. Analysing the implicit complexity of programs. *Information and Computation*, 2000. A paraître.

[Mét88] D. Le Métayer. Ace : an automatic complexity evaluator. *Transactions on programming languages and systems*, 10(2), 1988.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[MM00a] M. Margenstern and K. Morita. NP problems are tractable in the space of cellular automata in the hyperbolic plane. *Theoretical Computer Science*, 2000. A paraître.

[MM00b] J-Y Marion and J-Y Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.

[MN70] Z. Manna and S. Ness. On the termination of Markov algorithms. In *Third hawaii international conference on system science*, pages 789–792, 1970.

[Mül74] H. Müller. *Klassifizierungen der primitiv rekursivezn funktionen*. PhD thesis, Universität Münster, 1974.

[Nel86] E. Nelson. *Predicative Arithmetic*. Princeton University Press, Princeton, 1986.

[Nig98] K.-H. Niggl. The μ-measure as a tool for classifying computational complexity. *The bulletin of symbolic logic*, 4(1) :100–101, 1998.

[Nig00] K.-H. Niggl. The μ-measure as a tool for classifying computational complexity. *Archive for Mathematical Logic*, 2000. A paraître.

[NPS90] B. Nordström, K. Peterson, and J. Smith. *Programming in Martin-Löf's type theory.* Oxford, 1990.

[Oit97] I. Oitavem. New reursive characterization of the elementary functions and the functions computable in polynomial space. *Revista Matemática de la universidad complutense de Madrid*, 10(1), 1997.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Par92] M. Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94 :335–356, 1992.

[Pét66] R. Péter. *Rekursive Funktionen.* Akadémiai Kiadó, Budapest, 1966. English translation : *Recursive Functions*, Academic Press, New York, 1967.

[Pla78] D. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report R-78-943, Department of Computer Science, University of Illinois, 1978.

[Pra65] D. Prawitz. *Natural Deduction.* Almqvist and Wiskel, Uppsala, 1965.

[PSS81] W.J. Paul, J.I. Seiferas, and J. Simon. An information theoretic approach to time bounds for on-line computation. *Journal of Computer System Science*, 23(2) :108–126, 1981.

[Rit63] R. Ritchie. Classes of predictably computable functions. *Transaction of the American Mathematical Society*, 106 :139–173, 1963.

[Rit65] R. Ritchie. Classes of recursive functions based on Ackermann's function. *Pacific journal of mathematics*, 15(3), 1965.

[Ros84] H.E. Rose. *Subrecursion.* Oxford university press, 1984.

[Rov99] L. Roversi. A p-time completeness proof for light logics. In *Computer Science Logic, 13th International Workshop, CSL '99*, volume 1683 of *Lecture Notes in Computer Science*, pages 469–483, 1999.

[Rus87] M. Rusinowitch. Path of subterms ordering and recursive decomposition ordering revisited. *Journal of Symbolic Computation*, 3 :117–131, 1987.

[Sav98] J. Savage. *Models of computation, Exploring the power of computing.* Addison Wesley, 1998.

[Saz80] V. Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, 7 :319–323, 1980.

[Sch] H. Schwichtenberg. Feasible programs from proofs. http ://www.mathematik.uni-muenchen.de/∼schwicht/.

[Sch69] H. Schwichtenberg. Rekursionszahlen und die Grzegorczyk-hierarchie. *Archive for Mathematical Logic*, 12, 69.

[Sie97] W. Sieg. Step by recursive step : Church's analysis of effective calculability. *The bulletin of symbolic logic*, 3(2) :154–180, Jun 1997.

[Sim88] H. Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27 :177–188, 1988.

[Sim00] H. Simmons. *Derivation and Computation*, volume 51 of *Tracts in theoretical computer science*. Cambridge, 2000.

[SS63] J.C. Shepherdson and H.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2) :217–255, 1963.

[TG95] A. Turing and J.-Y. Girard. *La machine de Turing*. Seuil, 1995.

[Tur36] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Mathematical Society*, 42(2) :230–265, 1936. Traduction [TG95].

[Val96] P. Valarcher. Contribution à l'étude du comportement des algorithmes : le cas de la récursion primitive., Janvier 1996. Thése de doctorat.

[Wei95] A. Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139 :335–362, 1995.