Complexité implicite des calculs : interprétation de programmes

Guillaume Bonfante

To cite this version:

Guillaume Bonfante. Complexité implicite des calculs : interprétation de programmes. Complexité [cs.CC]. Institut National Polytechnique de Lorraine - INPL, 2011. tel-00656766

HAL Id: tel-00656766

https://theses.hal.science/tel-00656766

Submitted on 5 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Complexité implicite des calculs :
interprétation de programmes

MÉMOIRE

présenté et soutenu publiquement le 9 décembre 2011

pour l'obtention de l'

Habilitation à Diriger des Recherches

par

Guillaume Bonfante

Composition du jury

Rapporteurs : Roberto Amadio, Professeur, Université de Paris VII
Etienne Grandjean, Professeur, Université de Caen Basse-Normandie
Simona Ronchi della Rocha, Professeur, Université de Turin

Examinateurs : Jean-Yves Marion, Professeur, Université de Lorraine
Patrick Baillot, Chargé de recherches, ENS Lyon
Philippe de Groote, Directeur de recherches, INRIA
Claude Kirchner, Directeur de recherches, INRIA
Daniel Leivant, Professeur, Université d'Indiana

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

Mis en page avec la classe thloria.

Remerciements

All !.

Table des matières

1 Introduction **1**

1.1 Position du problème . . . 1

1.2 Et pour la suite, trois possibilités . . . 4

1.2.1 Extension du domaine des programmes . . . 4

1.2.2 L'application des interprétations, tout un programme . . . 6

1.2.3 Une théorie algorithmique de la réécriture . . . 7

Bibliographie **9**

Bibliographie **11**

2 Rewriting, a model of computation **13**

2.1 Term algebras . . . 13

2.1.1 Substitutions . . . 14

2.1.2 Contexts, extensions of contexts . . . 14

2.2 First Order Rewriting . . . 15

2.2.1 First order functional programs . . . 16

2.2.2 Semantics . . . 17

2.2.3 Rank, precedence and call-trees . . . 18

2.2.4 Computation by rewriting . . . 19

2.3 Termination . . . 19

2.3.1 Recursive Path ordering . . . 19

2.4 Interpretations of programs . . . 21

3 First results in Implicit computational complexity **25**

3.1 Interpretation over **N** . . . 27

3.1.1 A tiering of interpretations . . . 27

3.2 Polynomial interpretations . . . 29

3.3 Sup-interpretation . . . 30

4 Small complexity classes **35**
4.1 Linear Space . . . 35
4.2 Cons-free programs and LOGSPACE . . . 35
4.3 Alogtime . . . 37

5 Confluence from a complexity point of view **39**
5.1 Semantics . . . 40
5.2 `F`.n-program with polynomial interpretation . . . 41
5.3 `F`.n-program with polynomial quasi-interpretation . . . 41
5.4 Non confluent constructor preserving programs . . . 44

6 Interpretations over the reals **47**
6.1 Synthesis and Verification of interpretations over the reals . . . 47
6.2 Positivstellensatz and applications . . . 48
6.3 The role of reals in complexity . . . 50
6.4 Dependency Pairs with polynomial interpretation over the reals . . . 52

7 From Term Rewriting to polygraphs **55**
7.1 A first glance at polygraphs . . . 55
7.2 Polygraphic programs . . . 57
7.3 Polygraphic interpretations . . . 59
7.4 Complexity of polygraphic programs . . . 62
7.5 Cartesian polygraphic interpretations and the size of structure computations . . . 63
7.6 The size of computations . . . 65
7.7 Polygraphic programs and polynomial-time functions . . . 65

Bibliographie **67**

Chapitre 1

Introduction

1.1 Position du problème

Calculer la complexité d'un programme est une opération intrinsèquement difficile. Le premier souci vient de la définition même de la complexité. Mais même si l'on se met d'accord sur cette notion, que l'on emploie la machine de Turing comme modèle de référence ou que l'on travaille de manière plus abstraite comme peuvent le faire Moschovakis [114], ou Blum [25], on ne peut pas fournir de solution définitive à ce problème du fait de son indécidabilité notoire.

S'il y a un intérêt pratique évident à calculer la complexité des programmes— on parle aussi d'analyse des ressources, ce n'est pas le seul intérêt que la notion de complexité puisse avoir. Elle traduit certaines propriétés mathématiques des programmes, elle les classifie. C'est donc un moyen de "mesurer" les programmes ou les langages de programmation afférents.

L'étude que nous proposons s'inscrit dans le cadre de la complexité implicite des calculs. Selon Daniel Leivant [101], il s'agit de donner des caractérisations de la complexité sans faire de référence explicite à un modèle de calcul. En procédant de la sorte, on renforce la notion même de classe de complexité (qui devient moins arbitraire), on renouvelle les questions autour de la séparation des classes, et l'on créé un lien avec les méthodes formelles, dans le cadre du développement logiciel. Notons ici que notre approche se place dans le cadre de l'analyse statique des programmes : on calcule la complexité des programmes sans les exécuter. En particulier, l'approche n'est pas compatible avec le "monitoring" où l'on observe le programme en cours d'exécution.

On peut distinguer trois branches en complexité implicite des calculs : celle fondée sur les modèles finis, celle s'appuyant sur la logique et l'isomorphisme de Curry-Howard et enfin celle traitant de la récursion et plus généralement les programmes.

Si on se restreint aux domaines finis, Gurevich montre dans [75] que les fonctions définies par récursion primitive correspondent aux fonctions calculables dans Logspace (voir également Sazonov [129]). Il montre aussi que les fonctions récursives correspondent dans ce cadre aux fonctions de Ptime. Une autre approche pour construire de nouveaux prédicats (les fonctions globales de Gurevich) à partir de structures finies consiste à enrichir la logique du premier ordre par de nouveaux quantificateurs (plus petits points fixes, clôture de relations, etc). Immerman dans [85] et Vardi dans [137] ont ainsi proposé des caractérisations de Logspace et Ptime. Ces techniques ont été employées pour la modélisation et l'étude des bases de données, et de leurs langages de requêtes. Une autre forme de ce type de recherches est illustrée par l'article de Grandjean et Schwentick [69] qui considère les données sous la forme de "RAM data-structure". Les auteurs décrivent un (des) schéma de récursion (Linear Recursion Scheme) qui caractérise le

temps linéaire (sur RAM).

Un autre secteur de recherche, dynamique, de la complexité implicite concerne la logique. Il s'appuie sur la correspondance de Curry-Howard qui fait le lien entre les preuves et les programmes (écrits dans un langage fonctionnel, le λ-calcul). L'évaluation du λ-terme (le programme en cours d'évaluation) se lit comme l'élimination des coupures de la preuve associée. La logique linéaire introduite par Girard [64], parce qu'elle décompose finement les règles des preuves, est un cadre idéal pour l'étude de la complexité de tels programmes. En restreignant les règles de la logique, on peut ainsi caractériser des classes de complexité comme Ptime, Pspace, le temps élémentaire, etc. Un des travaux fondateurs dans ce domaine est celui proposé par Girard [65] dans lequel il introduit la logique linéaire "light". Elle donne une caractérisation de Ptime. Dans cette veine, Lafont [97] a défini la Soft Linear Logic qui caractérise le temps polynomial, Hofmann a décrit les fonctions non-size increasing dans [80, 81], puis Baillot et Terui [16, 17] définissent des types pour un λ-calcul correspondant au temps polynomial. D'autres classes de complexité ont été caractérisées, comme Pspace par Gaboardi, Marion et Ronchi dans [59], et NPtime par les mêmes auteurs [60].

Une troisième approche est issue de l'étude de la récursivité et, de manière connexe, des programmes fonctionnels du premier ordre. Les travaux que nous présentons ici s'inscrivent dans cette lignée. En restreignant la syntaxe, en imposant des contraintes sur les programmes, on peut caractériser les fonctions calculées par lesdits programmes en termes de complexité. Un des pionniers dans le domaine est Cobham qui donne une caractérisation du temps polynomial [47] en utilisant une borne a priori sur le taux de croissance des fonctions calculées. Par la suite, la notion de stratification des données des programmes que l'on trouve chez Bellantoni et Cook [20] et chez Leivant [100] a indiscutablement rénové la question de la complexité implicite dans le cadre de la théorie de la récursion. Un autre apport majeur dû à Leivant et Marion concerne la notion de ramification des programmes, voir [102], qui permet de décrire les programmes calculables en espace polynomial. La notion est utilisée pour caractériser ALogTime dans [103]. En essence, on retrouvera le même argument dans notre caractérisation de l'espace polynomial avec les quasi-interprétations [38] ou pour la caractérisation de NC_1 que l'on trouve dans [40].

L'étude des schémas de récurrence est un principe fécond pour l'étude de la complexité des programmes en général. En considérant les schémas de récurrence comme un modèle restreint de langage de programmation, on peut en extraire les principes fondamentaux, et les généraliser. Par exemple, les règles de la récurrence peuvent être traduites par des restrictions des motifs des règles de calcul pour la programmation fonctionnelle au premier ordre. C'est ce type de démarche qui a été adopté dans [40] où l'on décrit la classe NC_1 comme une forme “dérivée” de l'article [103]. Un autre exemple concerne les travaux réalisés autour des classes de complexité NC_k dans [35]. Ces résultats préliminaires sont maintenant prêts à être reconsidérés dans un cadre plus général de programmation fonctionnelle.

Un autre angle fructueux de recherche est l'analyse des preuves de terminaison en réécriture. Parce que ces preuves forment des moules qui contraignent la programmation, c'est un bon moyen d'obtenir des caractérisations implicites des programmes [1]. Nous reviendrons en détail sur l'étude des preuves de terminaison par interprétation de programmes fonctionnels du premier ordre.

En complexité implicite des calculs, il y a deux critères généraux qui permettent d'évaluer (au moins partiellement) une théorie, l'intentionnalité de la théorie, et son caractère effectif. Une fonction est calculée par plusieurs programmes (doit-on dire algorithme ?). L'intentionnalité mesure la quantité de programmes par fonction plutôt que la quantité de fonctions. A contrario, les caractérisations extentionnnelles mettent l'accent sur les fonctions, cela correspond à l'approche

1. Notons que la terminaison est une première caractérisation en termes de complexité d'un programme.

classique de la théorie de la complexité. Un résultat remarquable pour illustrer l'intentionnalité est celui de Colson [48] qui montre que l'on ne peut pas programmer le calcul du minimum de deux entiers m et n à l'aide de la récursion primitive en temps $O(\min(m, n))$, calcul que l'on peut faire avec la réécriture. On peut donc affirmer que la récursion primitive a une intentionnalité plus faible que la réécriture.

D'un point de vue pratique, une théorie avec une bonne intentionnalité est une théorie dans laquelle il est facile de programmer : il y a plus de "chances" de trouver le bon programme parce qu'il y en a "beaucoup". À titre d'illustration, le calcul de la fonction minimum dans la théorie de Bellantoni et Cook [20] est bien plus difficile qu'en programmation fonctionnelle du premier ordre. Les caractérisations de complexité implicite mettent très souvent en œuvre un critère permettant de classifier les programmes. De tels critères réduisent l'intentionnalité (en éliminant les programmes qui n'y répondent pas) mais donnent des informations sur la complexité de l'algorithme. Un jeu s'établit entre les critères et l'intentionnalité : plus les critères sont restrictifs, plus le contrôle sur les programmes est élevé, mais ceci se fait au détriment du nombre de programmes pris en compte.

Le deuxième point concerne l'aspect effectif de la théorie. La caractérisation la plus précise qu'on puisse faire d'une classe de complexité C consiste à dire qu'il s'agit de tous les programmes dont la complexité est C. Mais, sauf cas dégénéré, on ne peut pas décider si un programme donné appartient à la classe C. Le caractère effectif de la théorie se présente sous deux formes. Premièrement, étant donné un programme, peut-on dire qu'il est couvert par la théorie ? Deuxièmement, peut-on associer au programme un certificat (comme on associe un type) de correction du programme (relativement à la théorie) qui puisse être vérifié ? Le deuxième cas permet d'envisager le scénario où un certificat est associé à son programme, et l'utilisateur du programme peut alors vérifier que celui-ci ne lui prendra pas trop de ressources de calcul.

Les travaux présentés se sont articulés à partir de l'étude initiale [29] dans laquelle nous montrons comment les preuves de terminaison par interprétation polynomiale caractérisent les fonctions calculées en temps polynomial. Brièvement, une interprétation d'un programme est une fonction des configurations (en cours d'exécution) vers un ensemble ordonné. À chaque étape de calcul, l'interprétation des configurations diminue (relativement à l'ordre). Les fonctions calculées par des systèmes de réécriture avec constructeurs, admettant une interprétation polynomiale additive sur les entiers naturels, peuvent être calculées en temps polynomial, et inversement, toute fonction calculable en temps polynomial peut être calculée par un tel système. Partant de là, les développements ont été de deux ordres, premièrement se dégager des problématiques liées à la terminaison pour arriver à des outils adaptés à la complexité, et deuxièmement de prendre en compte les deux critères généraux concernant la complexité implicite des calculs.

Si l'on reprend [29], plusieurs hypothèses sont faites :

- la réécriture est employée comme un modèle de calcul, et en particulier, aucune notion de stratégie d'évaluation n'est considérée,
- le critère de complexité est un critère de terminaison (en outre, terminaison par interprétation),
- les fonctions employées pour les interprétations sont dans une classe restreinte de fonctions, les polynômes à coefficient entiers.

Nous allons voir en quoi ces trois paramètres, le modèle de calcul, la nature des critères, et les paramètres de ces critères peuvent être généralisés. Par exemple, l'étude des fonctions employées pour les interprétations a été réalisée dans [30, 37]. Nous y montrons que le critère fondamental concernant les fonctions (utilisées pour les interprétations) reste leur taux d'accroissement, de manière analogue à ce que l'on trouve pour les hiérarchies sous-récursives de fonctions [128, 132]. Mais le cadre de travail nous permet de distinguer des classes de complexité plus faibles comme

linspace et Logspace (cf. [37]). La forme des interprétations (alliées à d'autres considérations syntaxiques sur les programmes) a été également employée pour donner une caractérisation d'ALogTime, les machines alternantes fonctionnant en temps logarithmique (cf. [40]). Ces travaux ont été poursuivis par Marion et Péchoux dans [110] qui décrit la classes des NC_k pour $k \geq 1$.

Un deuxième point concerne la notion d'interprétation. Dans [36], la notion de quasi-interprétation a été introduite. Plus générale que la notion d'interprétation, au prix de la perte de la terminaison, elle permet de rendre compte de la taille des termes obtenus en cours de calcul. Techniquement, on remplace pour chaque étape de calcul une inégalité stricte par une inégalité large. Si on allie une quasi-interprétation avec une preuve de terminaison par ordre récursif sur les chemins, on obtient alors deux caractérisations, une de Ptime avec l'ordre multi-set, une de Pspace avec l'ordre lexicographique (cf. [38]). Ces résultats ont été repris par la suite, par exemple dans [111] en généralisant encore la notion de quasi-interprétation : les sup-interprétations.

Le troisième point concerne le modèle de calcul lui-même. Déjà, dans [36], il nous a fallu redéfinir la sémantique opérationnelle pour une évaluation efficace des programmes. La technique, dite de mémoisation, introduite par Jones dans [88], consiste essentiellement à se souvenir des appels récursifs pour éviter de refaire les mêmes calculs. Jones emploie cette technique pour montrer que les programmes “cons-free” calculent exactement les prédicats de Ptime. Nous avons repris ces travaux dans [27] où nous montrons que l'ajout d'un opérateur de choix pour les programmes “cons-free” ne modifie pas la caractérisation. Nous en tirons une caractérisation avec une meilleure intentionnalité. Mais c'est l'étude des programmes polygraphiques [33] qui nous offre le meilleur exemple de telle étude. Nous montrons qu'en employant des graphes plutôt que des termes, nous augmentons l'intentionnalité de la théorie. En outre, la distinction entre les invariants de terminaison, et les invariants géométriques (comme le font les quasi-interprétations) est beaucoup plus claire dans ce cadre.

1.2 Et pour la suite, trois possibilités

À la lumière des travaux passés, il me semble qu'il y a trois directions de recherches prometteuses. La première concerne l'amélioration générale des critères employés pour les caractérisations implicites. La deuxième, les aspects pratiques de ces méthodes, et en particulier, leur emploi dans un cadre plus standard que la programmation fonctionnelle du premier ordre. La troisième consiste à abstraire les résultats courants, pour une théorie générale de la complexité en réécriture.

1.2.1 Extension du domaine des programmes

Les deux principes moteurs que nous avons employés jusque là, l'étude des preuves de terminaison en réécriture, et l'étude des schémas de récursion, sont toujours actifs : de nouvelles méthodes de preuves de terminaison sont proposées en permanence (voir le concours annuel d'outils de preuve de terminaison organisé par l'Université d'Innsbruck [2]).

Bien sur, les problèmes de terminaison sont différents des problèmes de complexité. Par exemple, les questions concernant la commutativité, l'associativité, la distributivité, etc., proviennent typiquement de considérations algébriques, de questions de représentation des structures mathématiques. Dans ce cas, les aspects calculatoires ne sont pas (nécessairement) au cœur du problème.

2. http ://dev.aspsimon.org/projects/termcomp/

Deuxième point, la réécriture est un cadre plus général que celui dont nous avons besoin. Par exemple, nous nous restreignons aux systèmes constructeurs et (dans la plupart des cas) confluents. Ceci vient du fait que nous avons concentré notre analyse sur les programmes fonctionnels du premier ordre, programmes pour lesquels nous pouvons faire ces deux hypothèses. La plus grande généralité de la réécriture—vis-à-vis de la programmation fonctionnelle— a deux faces : une méthode s'appliquant aux systèmes non-constructeurs peut aller au delà de l'analyse des algorithmes que nous prenons en compte actuellement, elle peut être également vue comme une généralisation d'une catégorie d'algorithmes, par exemple ceux utilisant l'ordre supérieur. C'est le cas, typiquement, quand une règle filtre sur un pattern qui contient une fonction comme dans la règle de réécriture $\mathtt{map}(g(x), \mathbf{cons}(a, l)) \rightarrow \mathbf{cons}(g(a), \mathtt{map}(g(x), l))$. De manière plus générale, la terminaison des systèmes à l'ordre supérieur a été l'objet de nombreuses études ; nous citons essentiellement [24], parce que Blanqui, Jouannaud et Rubio généralisent l'ordre RPO que nous employons (au premier ordre), mais il y a d'autres références, comme van Raamsdonk [127]. Dans une autre veine, le ρ-calcul de Cirstea et C. Kirchner présenté dans [45, 46] a été conçu pour traiter de manière uniforme le λ-calcul et la réécriture.

En outre, parmi les nouvelles méthodes de preuves de terminaison, certaines prennent en compte la notion de stratégie de réduction. Par exemple, Gnaedig et Kirchner développent dans [66] une méthode de terminaison qui s'applique à de nombreuses stratégies (innermost, outermost, etc). D'autres travaux représentatifs de ce champ d'étude pour la stratégie innermost (que nous emploierons) sont [10, 63, 2]. L'introduction des stratégies permet de voir la réécriture, non comme une méthode de spécification, mais comme un langage de programmation. Une étude directe de la complexité peut-être alors réalisée pour ces langages.

À l'intérieur même du domaine des interprétations de programmes, de nouvelles méthodes doivent être étudiées. Hofbauer a proposé les interprétations dépendantes du contexte (context dependent interpretation) dans [78]. Elles généralisent la notion d'interprétation. Un des bénéfices de ce type d'interprétation est d'éviter la contrainte de sous-terme, une contrainte majeure des interprétations strictes et des quasi-interprétations. Une étude de la longueur de dérivation de tels systèmes a été réalisée par Moser [116]. Toutefois, elle introduit des contraintes très fortes sur l'ensemble des fonctions utilisées pour les interprétations. L'autre manière de généraliser les interprétations consiste à choisir un univers d'interprétation plus général que les entiers naturels. Un exemple de tel développement est donné, ici encore, par Hofbauer qui emploie l'espace des matrices [79]. Moser en a étudié également la complexité [117]. Il montre les liens entre ce type d'interprétation et les interprétations dépendantes du contexte.

Un autre principe moteur d'étude que nous avons déjà évoqué concerne l'étude des schémas de récursion. Nous avons proposé avec Reinhard Kahle, Jean-Yves Marion et Isabel Oitavem dans [35] une théorie pour décrire les classes de complexité de la hiérarchie des NC_k. Cette théorie raffine celle proposée par Daniel Leivant dans [101]. Ces résultats devraient être suivis d'une caractérisation de la hiérarchie imbriquée des AC_k pour laquelle nous avons dégagé les éléments essentiels. Ces travaux sont préparatoires pour des caractérisations employant un langage de programmation plus général. Le travail principal est alors de transcrire sous la forme de contraintes sur les interprétations, les contraintes syntaxiques dues aux schémas de récursion. Un exemple de telle extension apparaît dans [40], mais il me semble qu'on devrait faire une étude plus systématique de ces transports de structure. Il me semble qu'il y a, ici, un point de convergence possible avec le domaine des modèles finis, en particulier pour les classes de petite complexité. Grandjean a montré que la classe des problèmes résolus en temps linéaire sur machine RAM est assez robuste (cf. [68]). Il décrit également la classe `NLin`, le temps linéaire non déterministe. En indiquant (cf. [69]) que leurs caractérisations algébriques du temps linéaire (déterministe ou non) peut servir pour en donner une caractérisation logique, Grandjean et Schwentick font le

lien avec la théorie des jeux. Un lien qui a également été fait par Hofmann et Schöpp pour l'espace logarithmique dans [82, 130]. Ils introduisent une stratégie d'évaluation efficace (en espace logarithmique) qui prend la forme d'un dialogue joueur/opposant. Mais pour revenir aux classes en temps linéaire, Jones en a établi dans [87] une stratification à l'aide de son langage impératif sur les arbres binaires. Dans son livre [88], il fait le lien entre ce langage et la machine RAM.

1.2.2 L'application des interprétations, tout un programme

On pourrait se dire que les aspects pratiques de l'application de ce type de théorie ne sont qu'une affaire de bonne ingénierie. Il me semble toutefois que certaines questions restent d'ordre scientifique, et j'y vois deux raisons. Premièrement, le calcul des interprétations reste un problème ouvert. Le calcul des coefficients (réels) des polynômes (à variables réelles) est a priori exponentiel relativement à la taille du programme, et doublement exponentiel relativement au degré des polynômes. Deuxièmement, le programme soumis à l'analyse de complexité n'est pas écrit en langage machine, mais dans un langage de haut niveau. Il faut donc assurer l'adéquation entre l'analyse réalisée sur le programme et le coût "réel" de l'exécution du programme. En d'autres termes, il faut assurer une compilation qui reste honnête relativement à notre analyse.

À propos de la synthèse et de la vérification des interprétations, l'emploi des réels à la place des entiers demeure délicat. Nous n'avons que des bornes supérieures sur la complexité des algorithmes, algorithme qui est en temps exponentiel. Pour le logiciel CROCUS, nous avons choisi de tirer les coefficients (entiers) au hasard, la solution paraît en pratique plus efficace. Néanmoins, laisser le calcul de la synthèse aux fruits du hasard n'est pas non plus très satisfaisant. Pour moi, le problème reste donc ouvert. Il a été abordé par Shkaravska, van Eekelen et van Kesteren pour l'évaluation de la taille des formes normales des calculs (par exemple [131]) [3]. La proposition de Moser [117] à propos des interprétations dépendantes du contextes, malgré toutes ses qualités, reste embryonnaire. Elle se focalise sur une forme très particulière d'interprétation. On pourra retrouver, dans son habilitation [115], une vue plus générale de ces travaux.

Il me semble qu'il y a une autre approche pour aborder le problème de la synthèse. Elle vient de deux observations qui m'intriguent. Premièrement, le logiciel CROCUS fonctionne comme s'il était "déterministe". Rappelons que le cœur de l'algorithme consiste à vérifier des inégalités de polynômes tirés au hasard. Il se trouve que le nombre de tirages nécessaires pour trouver la solution d'un programme est remarquablement stable. Cela témoigne-t-il d'une structure sous-jacente ? Deuxièmement, lorsque l'on trouve une solution, on en trouve de nombreuses autres en changeant un certain nombre de paramètres. Cette observation apparaît déjà implicitement chez Amadio dans [6] quand il définit des systèmes d'équations linéaires pour déterminer les coefficients des solutions homogènes. Il me semble que pour paramétrer les équations des polynômes d'interprétation, une solution élégante consiste à reconsidérer le problème à l'ordre supérieur. Entre les deux schémas (pour le calcul de la multiplication et pour le codage d'Hufman) :

$$\texttt{mult}(\texttt{s}(x), y) \to \texttt{add}(y, \texttt{mult}(x, y))$$

et

$$\texttt{encode}(t, ::(a, m)) \to \texttt{concat}(\texttt{code}(t, a), \texttt{encode}(t, m))$$

on retrouve les mêmes équations pour les interprétations de `mult` et `encode` (à des paramètres près). D'ailleurs, les deux fonctions `mult` et `encode` peuvent être interprêtées par la même fonction $(\!|\texttt{encode}|\!)(x, y) = (\!|\texttt{mult}|\!)(x, y) = x \times y + x$. De ce point de vue, pour calculer une interprétation,

3. A ceci près que l'objectif de cette étude est une évaluation exacte des tailles des résultats.

il “suffit” de retrouver le motif de la récursion (relativement au contexte des interprétations déjà présentes). Ces motifs peuvent être modélisés à l’ordre supérieur. Le rapprochement entre la synthèse et l’ordre supérieur (mais dans l’autre direction) a déjà été opéré dans [41], où la stratification modulaire des programmes (qui simplifie la recherche d’interprétations) rend compte des programmes écrits à l’ordre supérieur après défonctionnalisation (cf. [53]).

L’autre point problématique de l’approche actuelle est de se restreindre à la programmation fonctionnelle (en outre, au premier ordre). Il est bien difficile dans ce cadre de tenir compte des aspects impératifs de la programmation. Bien sûr, la complexité implicite ne s’est pas arrêtée à la programmation fonctionnelle. Le travail remarquable de Jones [89] établit une hiérarchie de (caractérisation de) classes de complexité pour un langage impératif (affectations et boucles sans contraintes) employant les arbres comme structure de donnée. Il n’est pas le seul. Niggl, par exemple, a étudié intensément ces aspects, en donnant de nombreux critères sur l’imbrication des boucles [122, 123], pour caractériser la complexité des programmes impératifs. Autour de Kristiansen et de Jones, les travaux autour des interprétations avec des matrices “mwp”, se rapprochent de nos méthodes [95, 90]. Marion et Moyen ont également abordé le problème, en employant les graphes de contrôle de ressources –une forme de réseau de Petri– comme abstraction des calculs dans [108, 118]. Le lien entre ces méthodes et les méthodes par interprétation reste à établir. À ce sujet, les travaux de Moyen et Metnani [119] nous semblent prometteurs. Plus proches d’une application standard, le système COSTA permet l’analyse de ressources pour un bytecode JAVA [4, 3]. Chez Microsoft, dans cette veine, nous mentionnons le travail de Gulwani [74].

Par ailleurs, il n’y a pas eu d’étude précise du lien entre la notion d’interprétation comme nous l’entendons et les interprétations abstraites, à la manière de Cousot (Patrick et Radhia), cf. e.g. [51]. Une interprétation à notre sens est bien une approximation de la taille des données calculées (approximation par le haut, c’est-à-dire dans un intervalle $[0, (\!| t |\!)]$ pour un terme t). La stabilité pour les règles et les contextes traduit l’invariance de l’approximation lors du calcul. Nous devrions être relativement proches de la connexion entre les deux notions. Et j’y vois un accès envisageable à la programmation impérative.

Enfin, pour le passage d’un langage de haut niveau à sa compilation dans un langage de bas niveau, toujours dans le cadre de la complexité implicite des calculs, je retiendrai deux approches, celle qu’on peut trouver chez Amadio et al dans [7], et la compilation de circuits booléens, comme dans [113] (ou comme dans Bellantoni et Oitavem [21], ou comme dans Bonfante et al [34, 35]). Revenons à [7], les auteurs décrivent un langage impératif de bas niveau, et, à l’aide de quasi-interprétations, ils assurent l’évaluation de la complexité des programmes afférents. Le procédé consiste à transporter les interprétations obtenues dans un langage fonctionnel vers le langage de bas niveau, employant une pile pour les appels de fonctions.

1.2.3 Une théorie algorithmique de la réécriture

Vue comme un modèle de calcul, la réécriture est Turing-complète, et comme chaque étape de calcul peut être simulée en temps constant sur une machine de Turing (cf. Martini et dal Lago [52]), la longueur de dérivation est en fait une bonne mesure de complexité. Le souci vient du fait qu’un modèle de calcul n’a pas forcément une notion intrinsèque de coût, du moins une notion compatible avec les autres modèles de calculs. Martini et dal Lago proposent d’appeler ce type de recherche, la complexité implicite des calculs du point de vue micro ("in the small"). Il me semble que l’on devrait faire le pont avec les "Abstract State Machines" de Gurevich [76, 56], pour lesquels il y a une axiomatisation précise des capacités d’un modèle de calcul "raisonnable". Le problème peut se poser pour des modèles de calculs qui ne sont pas Turing-complets. Par

exemple, dans le cadre du traitement automatique des langues, nous avons montré l'utilité de la réécriture de graphe avec conservation des noeuds (cf. [32] pour le calcul de la sémantique vers la syntaxe. Faire le lien avec d'autres modèles de calculs permet alors d'envisager a) une procédure de compilation de nos programmes, mais surtout b) de comparer notre proposition avec d'autres méthodes (comme celle s'appuyant sur des lambda-calculs restreints).

L'autre souci vient de la profusion des caractérisations de classes de complexité. Rien qu'en se restreignant à la réécriture, on peut trouver une dizaine de critères pour décrire le temps polynomial ([29, 31, 38, 27, 12, 77, 15, 111, 44, 119]). Elles n'ont pas toute la même intentionalité, du moins informellement. Habituellement, pour comparer sa théorie avec les autres, chaque auteur met en avant un exemple particulièrement criant, sensé illustrer l'intentionalité de la théorie. Pour aller plus loin, comme la comparaison d'ensembles de programmes (syntaxes différentes, sémantiques différentes) est encore plus difficile que la comparaison d'ensembles de fonctions, nous proposons de les comparer de manière indirecte. L'idée est de transformer deux ensembles de programmes de façon à les distinguer extensionnellement. Nous avons montré ainsi dans [28] comment l'ajout de la non-confluence permet de séparer des classes de programmes. Ces travaux doivent être formalisés, et approfondis.

Un autre procédé pour réunifier toutes ces approches est de travailler de manière plus abstraite. Ce point de vue est bien illustré par le travail de Gaboardi et Redmond [61], dans le domaine de la logique, domaine qui s'apprête mieux à une approche catégorique. Bien sûr, je n'oublie pas l'article fondateur de Moschovakis "What is an algorithm" [114] qui me semble justifier cette démarche.

Ce dont nous ne parlerons pas

Les travaux présentés dans ce document ne recouvrent pas l'ensemble de mes recherches académiques. Néanmoins, ils sont peut-être ceux qui sont le plus aboutis, et sans doute les mieux reconnus. C'est pourquoi j'ai choisi de les exposer pour cette thèse d'habilitation à diriger des recherches. Néanmoins, il est incontestable que mon savoir faire en complexité des calculs a orienté largement mon approche des autres champs de recherches qui m'intéressent et au premier rang la virologie informatique. Mes travaux en traitement automatique des langues ont eux aussi bénéficié de ces connaissances : j'ai pu trouver là un terrain d'application aux notions algorithmiques comme la mémoisation, la programmation dynamique, etc. Voici une courte bibliographie de ces travaux annexes :

Bibliographie

[1] Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud, *A computability perspective on self-modifying programs*, Software Engineering and Formal Methods (Hanoi Viet Nam), 11 2009.

[2] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion, *An implementation of morphological malware detection*, EICAR (Laval France), 2008, pp. 49–62.

[3] ______, *Morphological Detection of Malware*, International Conference on Malicious and Unwanted Software MALWARE 2008 (Alexendria VA États-Unis d'Amérique), Fernando C. Colon Osorio, IEEE, 2008.

[4] ______, *Architecture of a morphological malware detector*, Journal in Computer Virology **5** (2009), no. 3, 263–270.

[5] ______, *A classification of viruses through recursion theorems*, CiE '07 : Proceedings of the 3rd conference on Computability in Europe (Berlin, Heidelberg), Springer-Verlag, 2007, pp. 73–82.

[6] ______, *On abstract computer virology from a recursion theoretic perspective*, Journal in Computer Virology **1** (2006), no. 3-4, 45–54.

[7] ______, *Toward an abstract computer virology*, ICTAC'05 (Hanoï/Vietnam) (Dang Van Hung and Martin Wirsing, eds.), Lecture Notes in Computer Science, vol. 3722, Springer, 2005, pp. 579–593.

[8] Guillaume Bonfante, Bruno Guillaume, Mathieu Morey and Guy Perrier, *Modular Graph Rewriting to Compute Semantics*, 9th International Conference on Computational Semantics (Oxford, UK), To be published in the conference proceedings.

[9] Guillaume Bonfante, Bruno Guillaume, and Mathieu Morey, *Dependency constraints for lexical disambiguation*, Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09) (Paris, France), Association for Computational Linguistics, October 2009, pp. 242–253.

[10] Guillaume Bonfante and Joseph Le Roux, *Intersection Optimization is NP-Complete*, Sixth International Workshop on Finite-State Methods and Natural Language Processing - FSMNLP 2007 (Postdam Allemagne), 2007.

[11] Guillaume Bonfante, Bruno Guillaume, and Guy Perrier, *Polarization and abstraction of grammatical formalisms as methods for lexical disambiguation*, COLING '04 (Morristown, NJ, USA), Association for Computational Linguistics, 2004.

[12] Guillaume Bonfante, Joseph Le Roux, and Guy Perrier, *Lexical disambiguation with polarities and automata*, CIAA (Oscar H. Ibarra and Hsu-Chun Yen, eds.), Lecture Notes in Computer Science, vol. 4094, Springer, 2006, pp. 283–284.

Notes sur la contribution technique

Place maintenant à la contribution technique de cette habilitation. Le chapitre 2 fixe le cadre de travail, la programmation du premier ordre avec 'matching'. On y définit également les ordres de terminaison que nous employons ensuite, et nous classifions les différentes notions d'interprétations que nous retrouverons. Le chapitre 3 présente les interprétations de programmes du point de vue de la complexité, et fait un bref rappel des premiers résultats obtenus, pour les interprétations et les quasi/sup-interprétations. Le chapitre 4 s'intéresse aux petites classes de complexité : LINSPACE, ALogTime et LOGSPACE. Nous montrons qu'elles peuvent être décrites par des restrictions supplémentaires sur la syntaxe et les interprétations. Le chapitre 5 envisage le cas des programmes non confluents. Dans le chapitre 6, nous montrons que les réels peuvent remplacer les entiers naturels sans dommage pour ce qui concerne la complexité. L'intérêt de ce choix vient du théorème de Tarski [135] qui permet d'envisager la synthèse automatique d'interprétations. Le chapitre 7 généralise le modèle de calcul, en considérant la réécriture de (poly)graphes plutôt que de termes.

Certains des résultats présentés ont été revus avec un peu de recul par rapport à ce qui a été initialement publié. Nous redonnons dans ce cas les preuves pour les cas non triviaux. Ci-dessous, une liste de mes travaux en complexité implicite des calculs.

Bibliographie

[1] Guillaume Bonfante, *Observation of implicit complexity by non confluence*, Electronic Proceedings in Theoretical Computer Science **23** (2010).

[2] Guillaume Bonfante and Florian Deloup, *Complexity invariance of real interpretations*, Theory and Applications of Models of Computation, 7th Annual Conference, TAMC 2010 (Jan Kratochvíl, Angsheng Li, Jirí Fiala, and Petr Kolman, eds.), LNCS, vol. 6108, Springer, 2010, pp. 139–150.

[3] Guillaume Bonfante and Georg Moser, *Characterising space complexity classes via knuth-bendix orders*, LPAR (Yogyakarta), 2010, pp. 142–156.

[4] Guillaume Bonfante and Yves Guiraud, *Polygraphic programs and polynomial-time functions*, Logical Methods in Computer Science (LMCS) **5** (2009), 1–37.

[5] Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem, *Recursion Schemata for NCk*, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings (Bertinoro Italie) (Michael Kaminski and Simone Martini, eds.), vol. 5213, Springer, 2008, pp. 49–63.

[6] Guillaume Bonfante and Yves Guiraud, *Intensional properties of polygraphs*, Proceedings of the 4th International Workshop on Term Graph Rewriting (TERMGRAPH 07), Electronic Notes in Computer Science, 2007.

[7] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen, *Quasi-interpretations : a way to control resources*, Theoretical Computer Science (2009), to appear.

[8] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux, *Quasi-interpretation Synthesis by Decomposition : An application to higher-order programs*, ICTAC (Macao Chine), 2007.

[9] Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem, *Towards an implicit characterization of NC^k*, Computer Science Logic '06 (Zoltán Èsik, ed.), Lecture Notes in Computer Science, vol. 4207, Springer, 2006, pp. 212–224.

[10] Guillaume Bonfante, *Some programming languages for LOGSPACE and PTIME*, 11th International Conference on Algebraic Methodology and Software Technology - AMAST'06 (Kuresaare/Estonie), 07 2006.

[11] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux, *A characterization of alternating log time by first order functional programs*, LPAR (Springer, ed.), Lecture Notes in Computer Science, vol. 4246, 2006, pp. 90–104.

[12] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen, *Quasi-Interpretations and Small Space Bounds*, Rewrite Techniques and Applications (J. Giesl, ed.), Lecture Notes in Computer Science, vol. 3467, Springer, April 2005, pp. 150–164.

[13] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux, *Synthesis of quasi-interpretations*, Logic and Complexity in Computer Science, 2005.

[14] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen, *On lexicographic termination ordering with space bound certifications*, Perspectives of System Informatics, PSI 2001, Novosibirsk, Russia (Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, eds.), Lecture Notes in Computer Science, vol. 2244, Springer, Jul 2001.

[15] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet, *Algorithms with polynomial interpretation termination proof*, J. Funct. Program. **11** (2001), no. 1, 33–53.

[16] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet, *Complexity classes and rewrite systems with polynomial interpretation*, Proceedings of the 12th International Workshop on Computer Science Logic (London, UK), Springer-Verlag, 1999, pp. 372–384.

Chapitre 2

Rewriting, a model of computation

We briefly recall the syntax and the semantics of rewriting. However, we suppose familiarity with this framework. We essentially use rewriting as a nice model of first order programs. Consequently, we will put the focus on constructor rewriting systems. We refer the reader to Dershowitz and Jouannaud's survey [57].

2.1 Term algebras

All along, we suppose that we are given a denumerable set $\mathcal{X}$ of *variables*.

A *signature* is a set of *symbols*, each symbol being given by its *name* and its *arity*. In the following, we suppose signatures to be finite. Symbols of arity 0 are *constants*, those of arity 1 are *unary* symbols, etc.

Given a signature Σ, we denote by $\mathcal{T}(\Sigma, \mathcal{X})$ the set of terms over the signature Σ and the variables $\mathcal{X}$. The set of *closed terms* is written $\mathcal{T}(\Sigma)$.

$$
\begin{array}{lll}
\textit{(Closed terms)} \quad \mathcal{T}(\Sigma) \ni s & ::= & c \mid f(s_1, \ldots, s_n) \\
\textit{(Terms)} \quad \mathcal{T}(\Sigma, \mathcal{X}) \ni t & ::= & x \mid c \mid f(t_1, \cdots, t_n).
\end{array}
$$

where c is a constant, x is a variable and f is a symbol of arity n.

To denote a lists of terms, we use the notation $\vec{t} = (t_1, \ldots, t_n)$. The length of the list is left implicit. Finally, given some unary constructor $\mathbf{c}$ and some constant a, we define

$$
\begin{array}{ll}
\mathbf{c}^0(a) & = a \\
\mathbf{c}^{n+1}(a) & = \mathbf{c}(\mathbf{c}^n(a))
\end{array}
$$

The *size* of a term t, written $|t|$, is defined by induction :

$$
\begin{array}{rcll}
|x| & = & 1 & x \in \mathcal{X} \\
|c| & = & 1 & c \text{ is a constant} \\
|f(t_1, \ldots, t_n)| & = & 1 + \sum_{i=1}^{n} |t_i|
\end{array}
$$

The *height* of a term is defined by :

$$
\begin{array}{rcll}
||x|| & = & 0 & x \in \mathcal{X} \\
||c|| & = & 0 & c \text{ is a constant} \\
||f(t_1, \ldots, t_n)|| & = & 1 + \max_{i=1}^{n} ||t_i||
\end{array}
$$

A position in a term is given by a string in $\mathbf{N}^*$. ε denotes the empty word. $t_{|p}$ denotes the sub-term at position p in t.

$$\begin{aligned} t_{|\varepsilon} &= t \\ f(t_1, \ldots, t_n)_{|i.p} &= t_{i|p} \end{aligned}$$

2.1.1 Substitutions

Given a term t, a variable x and a term u, $t[x \leftarrow u]$ denotes the replacement of x by u in t. Then, $t[x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n] = t[x_1 \leftarrow u_1][\cdots][x_n \leftarrow u_n]$ denotes the composition of such replacements. The reader should be aware of conflicts of variables for such multiple replacements. Actually, since we essentially replace variables by closed terms, this is not an issue for us.

A *substitution* σ is a finite mapping from variables to terms. It applies to terms through the equality $t\sigma = t[x_1 \leftarrow \sigma(x_1), \ldots, x_k \leftarrow \sigma(x_k)]$ where $x_1, \ldots, x_k$ is the domain of the substitution. Again, we will mainly consider *closed term substitution*, that is for all x in the domain of σ, the term $\sigma(x)$ is a closed term.

Let $\trianglelefteq$ and $\underline{\blacktriangleleft}$ denote respectively the *sub-term* and the *embedding* relations on terms. They are defined by :

$$\frac{}{t \trianglelefteq t} \qquad \frac{}{t \,\underline{\blacktriangleleft}\, t}$$

$$\frac{t \trianglelefteq u_i}{t \trianglelefteq f(u_1, \cdots, u_n)} \qquad \frac{t \,\underline{\blacktriangleleft}\, u_i}{t \,\underline{\blacktriangleleft}\, f(u_1, \cdots, u_n)}$$

$$\frac{\forall i : t_i \,\underline{\blacktriangleleft}\, u_i}{f(u_1, \cdots, u_n) \,\underline{\blacktriangleleft}\, f(t_1, \cdots, t_n)}$$

These two relations are partial orders. Let $\triangleleft$ and $\blacktriangleleft$ denote respectively their strict part.

2.1.2 Contexts, extensions of contexts

A *context* is a term t whose variables occur only once. $t[u_1, \ldots, u_n]$ is a shorthand for $t[x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n]$ where $x_1, \ldots, x_n$ are the variables of t given is some a priori order. Actually, from the context, there is no confusion about this order in the sequel. We will not mention it from now on.

We say that two contexts t_1 and t_2 are *compatible* ($t_1 \,|_{\mathrm{com}}\, t_2$) if either t_1 or t_2 is a variable or $t_1 = f(u_1, \cdots, u_n)$, $t_2 = f(v_1, \cdots, v_n)$ and for all $i \leq n$, $u_i \,|_{\mathrm{com}}\, v_i$.

The *extension of two compatible contexts* t_1 and t_2, denoted $t_1 + t_2$, is defined by :

$$\begin{aligned} x + t &= t \\ t + x &= t \\ f(u_1, \cdots, u_n) + f(v_1, \cdots, v_n) &= f(u_1 + v_1, \ldots, u_n + v_n) \end{aligned}$$

where x denotes a variable and f is a symbol.

Definition 1 (Compatible extension)**.** Given some terms t'_1, t_1, t_2 such that $t'_1 \trianglelefteq t_1$ and $t_1 |_{\text{com}}\, t_2$, the *compatible extension* of t'_1 w.r.t. t_2 is defined as the largest term t such that $t \trianglelefteq t_1 + t_2$ and $t |_{\text{com}}\, t'_1$. It is defined by the equations $\text{ext}(t'_1, t_1, t_2) =$

$$\begin{cases} t'_1 & \text{if } t_2 \text{ is a variable} \\ t_2 & \text{if } t_1 \text{ is a variable} \\ \text{ext}(t'_1, v_i, w_i) & \text{if } t_1 = \mathbf{c}(\vec{v}), t_2 = \mathbf{c}(\vec{w}) \\ & \text{ and } t'_1 \trianglelefteq v_i \\ \mathbf{c}(\text{ext}(u_1, v_1, w_1), \dots, \text{ext}(u_n, v_n, w_n)) & \text{if } t_1 = \mathbf{c}(\vec{v}), t_2 = \mathbf{c}(\vec{w}), t'_1 = \mathbf{c}(\vec{u}) \\ & \text{ and for all } i \leq n, u_i \trianglelefteq v_i \end{cases}$$

Proposition 2. *Given a term t and two contexts t_1, t_2 such that $t = t_1[u_1, \dots, u_n] = t_2[v_1, \dots, v_m]$, then t_1 and t_2 are compatible.*

Moreover, suppose that $t'_1 \trianglelefteq t_1$, let $x_{i_1}, \dots, x_{i_k}$ be the subset of variables in t'_1 occuring in t_1. There are positions $p_1, \dots, p_h$ in $v_1, \dots, v_m$ such that $t' = t'_1[u_{i_1}, \dots, u_{i_k}] = ext(t'_1, t_1, t_2)[v_{i_1 | p_1}, \dots, v_{i_h | p_h}]$.

Démonstration. By induction on the definitions. □

2.2 First Order Rewriting

A *rule* is pair of terms, written $\ell \to r$, such that variables occurring in r also occur in ℓ. Furthermore, ℓ is not a variable.

Definition 3. A *rewriting system* is given by $\langle \Sigma, \mathcal{R} \rangle$ where

1. Σ is a (finite) signature,
2. $\mathcal{R}$ is a finite set of rewriting rules.

A rewriting system $\langle \Sigma, R \rangle$ induces a relation $\to_R$ on terms by closure under substitution and context. When the context is clear, we remove the subscript R and use $\to$ instead of $\to_R$. The relation $\xrightarrow{*}$ is the reflexive and transitive closure of $\to$. A *normal form* t is a term which cannot be reduced, that is there is no u such that $t \to u$. We denote by $t \xrightarrow{!} v$ the fact that $t \xrightarrow{*} v$ and v is a normal form.

$t_0 \xrightarrow{n} t_n$ denotes the fact that $t_0 \to t_1 \cdots \to t_n$. The derivation height is defined for a term t as the maximal length of a derivation :

$$\mathtt{dh}(t) = \max\{n \in \mathbf{N} \mid \exists v : t \xrightarrow{n} v\}.$$

The derivational complexity is the function $\mathtt{dc} : \mathbf{N} \to \mathbf{N}$ with :

$$\mathtt{dc}(n) = \max\{\mathtt{dh}(t) \mid |t| \leq n\}.$$

We say that a term rewriting system is *confluent* if for all terms t, u and v such that $v \xrightarrow{*} u$ and $v \xrightarrow{*} t$, there is some w such that $u \xrightarrow{*} w$ and $t \xrightarrow{*} w$. Observe that for confluent rewriting systems, a term t has at most one normal form.

Throughout, to justify the fact that a term rewriting system is confluent, we use the criterium of orthogonality, a sufficient condition due to Huet [84]. A term rewriting system is *orthogonal* if it satisfies both conditions : (i) each rule $\ell \to r$ is left-linear, that is a variable appears only once in ℓ, and (ii) there are no two left hand-sides which are overlapping.

Example 4. Let us consider the signature $\Sigma = \{\mathbf{0}^0, \mathtt{s}^1, \mathtt{add}^2, \mathtt{mult}^2\}$ where the superscript denotes the arity of the symbol. We define the rules :

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

This famous rewriting system computes [4] the addition and the multiplication of tally integers. Notice that the rules define implicitly the set of symbols with their arity. Consequently, from now on, we present rewriting systems by their rules.

This following program sorts lists of tally numbers. To improve the readability, we use an infix notation.

Example 5. Given a list l of tally natural numbers, $\mathtt{sort}(l)$ sorts the elements of l by insertion.

$$\begin{aligned} \mathtt{if}\ \mathbf{tt}\ \mathtt{then}\ x\ \mathtt{else}\ y &\to x \\ \mathtt{if}\ \mathbf{ff}\ \mathtt{then}\ x\ \mathtt{else}\ y &\to y \\ \mathbf{0} < \mathtt{s}(y) &\to \mathbf{tt} \\ x < \mathbf{0} &\to \mathbf{ff} \\ \mathtt{s}(x) < \mathtt{s}(y) &\to x < y \\ \mathtt{insert}(a, \varepsilon) &\to \mathbf{cons}(a, \varepsilon) \\ \mathtt{insert}(a, \mathbf{cons}(b, l)) &\to \mathtt{if}\ a < b\ \mathtt{then}\ \mathbf{cons}(a, \mathbf{cons}(b, l)) \\ &\qquad \mathtt{else}\ \mathbf{cons}(b, \mathtt{insert}(a, l)) \\ \mathtt{sort}(\varepsilon) &\to \varepsilon \\ \mathtt{sort}(\mathbf{cons}(a, l)) &\to \mathtt{insert}(a, \mathtt{sort}(l)) \end{aligned}$$

2.2.1 First order functional programs

For us, a first order functional program can be seen as the particular case of rewriting system where the signature Σ splits into two disjoint sets of symbols, constructor symbols and function symbols. Such systems are also known as *constructor rewriting systems*. The hygiena for such rewriting systems is that left hand side of rules must be of the form $\mathtt{f}(p_1, \ldots, p_n)$ with $\mathtt{f}$ a function symbol and p_i terms over the constructor symbols, $i \in 1..n$.

Definition 6. A $\mathtt{F}.n$-program is a 4-tuple $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ such that

1. $\langle \mathcal{C} \cup \mathcal{F}, \mathcal{R} \rangle$ is a rewriting system,
2. $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{C}, \mathcal{X})$ is a term corresponding to the "main" expression, that is the expression to be evaluated,
3. for all rule $\ell \to r \in \mathcal{R}$, we have $\ell = \mathtt{f}(p_1, \ldots, p_n)$ with $\mathtt{f} \in \mathcal{F}$ and $p_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, $i = 1, \ldots, n$.

$\mathtt{F}.n$ is the set of such programs.

We also consider programs with confluent underlying rewriting systems :

Definition 7. A $\mathtt{F}$-program is a $\mathtt{F}.n$-program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ such that $\langle \mathcal{C} \cup \mathcal{F}, \mathcal{R} \rangle$ is a *confluent* rewriting system.

4. A more formal definition of the word "compute" will come soon.

2.2.2 Semantics

What is the function computed by a program ? Several semantics are conceivable. We use two of them, a naïve one, and a refinment of the naïve one where we distinguish inputs from outputs. Both make the link between a set of programs and a set of functions.

Let us present the naïve semantics of F-programs. The domain of the computations is the term algebra $\mathcal{T}(\mathcal{C})$. We say that a (partial) function $\varphi : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ is computed by a F-program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ if :

– t has exactly n variables written $x_1, \ldots, x_n$,
– for all $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C})$,

$$\varphi(t_1, \ldots, t_n) \text{ is defined} \Leftrightarrow t[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n] \xrightarrow{!} \varphi(t_1, \ldots, t_n).$$

Conversely, a F-program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ computes the partial function $[\![t]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ defined as follows. For every $t_1, \cdots, t_n \in \mathcal{T}(\mathcal{C})$, $[\![t]\!](t_1, \cdots, t_n) = v$ iff $t[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n] \xrightarrow{!} v$ and $v \in \mathcal{T}(\mathcal{C})$. Otherwise, it is undefined.

Example 8. Equality on binary words in $\{0,1\}^*$, boolean operations, membership in a list (built on **cons**, nil) are computed as follows. We add to the rules of Example 5,

$$\begin{aligned}
\varepsilon = \varepsilon &\to \mathbf{tt} \\
\mathbf{i}(x) = \mathbf{i}(y) &\to x = y \text{ with } \mathbf{i} \in \{0,1\} \\
\mathbf{i}(x) = \mathbf{j}(y) &\to \mathbf{ff} \text{ with } \mathbf{i} \neq \mathbf{j} \in \{0,1\} \\
\mathtt{not}(\mathbf{tt}) &\to \mathbf{ff} \\
\mathtt{not}(\mathbf{ff}) &\to \mathbf{tt} \\
\mathtt{or}(\mathbf{tt}, y) &\to \mathbf{tt} \\
\mathtt{or}(\mathbf{ff}, y) &\to y \\
\mathtt{and}(\mathbf{tt}, y) &\to y \\
\mathtt{and}(\mathbf{ff}, y) &\to \mathbf{ff} \\
\mathtt{in}(a, \mathtt{nil}) &\to \mathbf{ff} \\
\mathtt{in}(a, \mathbf{cons}(b, l)) &\to \mathtt{if}\ a = b\ \mathtt{then}\ \mathbf{tt}\ \mathtt{else}\ \mathtt{in}(a, l)
\end{aligned}$$

Example 9. Given two binary words u and v over the constructor set $\{\mathbf{a}, \mathbf{b}, \varepsilon\}$, $\mathtt{lcs}(u, v)$ returns the the length of the longest common subsequence of u and v. The expression $\mathtt{lcs}(\mathbf{ababa}, \mathbf{baaba})$ evaluates to $\mathbf{s}^4(\mathbf{0})$ that is the length of the longest common subsequence (take **baba**).

$$\begin{aligned}
\mathtt{max}(n, \mathbf{0}) &\to n \\
\mathtt{max}(\mathbf{0}, m) &\to m \\
\mathtt{max}(\mathbf{s}(n), \mathbf{s}(m)) &\to \mathbf{s}(\mathtt{max}(n, m)) \\
\mathtt{lcs}(\varepsilon, y) &\to \mathbf{0} \\
\mathtt{lcs}(x, \varepsilon) &\to \mathbf{0} \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{i}(y)) &\to \mathbf{s}(\mathtt{lcs}(x, y)) && \mathbf{i} \in \{\mathbf{a}, \mathbf{b}\} \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{j}(y)) &\to \mathtt{max}(\mathtt{lcs}(x, \mathbf{j}(y)), \mathtt{lcs}(\mathbf{i}(x), y)) && \mathbf{i} \neq \mathbf{j} \in \{\mathbf{a}, \mathbf{b}\}
\end{aligned}$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(v_1, \cdots, v_n)} \textit{(Constructor)}$$

$$\frac{\forall i, t_i \downarrow v_i \quad \mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{R} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad r\sigma \downarrow v}{\mathtt{f}(t_1, \cdots, t_n) \downarrow v} \textit{(Function)}$$

FIGURE 2.1 – Call by value semantics with respect to a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$.

Sometimes, we will distinguish input constructors from output constructors. We suppose given a signature A whose corresponding term algebra is the domain of the computations. A map $\alpha : A \to \mathcal{C}$ induces an homomorphism of terms by $\mathbf{c}(t_1, \ldots, t_n) \mapsto \alpha(\mathbf{c})(\alpha(t_1), \ldots, \alpha(t_n))$. We say that a function $\varphi : \mathcal{T}(A)^k \to \mathcal{T}(A)$ is computed by a F-program $\langle \mathcal{C}_{in} \cup \mathcal{C}_{out}, \mathcal{F}, t, \mathcal{R} \rangle$ with respect to the in-out semantics if

- t has exactly n variable written $x_1, \ldots, x_n$,
- there are two injective maps $\alpha : A \to \mathcal{C}_{in}$ and $\beta : A \to \mathcal{C}_{out}$ such that for all $t_1, \ldots, t_n \in \mathcal{T}(A)$,

$$\varphi(t_1, \ldots, t_n) \text{ is defined} \Leftrightarrow t[x_1 \leftarrow \alpha(t_1), \ldots, x_n \leftarrow \alpha(t_n)] \xrightarrow{!} \beta(\varphi(t_1, \ldots, t_n)).$$

Remark 10. The drawback of this latter semantics compared to the naive one is that we loose an internal notion of composition. Indeed, let us suppose that $\varphi : \mathcal{T}(\mathcal{C}) \to \mathcal{T}(\mathcal{C})$ and $\psi : \mathcal{T}(\mathcal{C}) \to \mathcal{T}(\mathcal{C})$ are respectively computed by $\langle \mathcal{C}, \mathcal{F}, t_\varphi, \mathcal{R} \rangle$ and $\langle \mathcal{C}, \mathcal{F}, t_\psi, \mathcal{R} \rangle$ according to the naive semantics. The function $\varphi \circ \psi$ is computed by $\langle \mathcal{C}, \mathcal{F}, t_\varphi[x \leftarrow t_\psi], \mathcal{R} \rangle$ where x is the only free variable of t_φ. Such a composition by substitution cannot be performed in general with the in-out semantics.

A more procedural semantics is the call-by-value semantics, displayed in Figure 2.1. The meaning of $t \downarrow v$ is that t evaluates to a constructor term v. $\mathfrak{S}$ denotes the set of constructor substitutions, that is the set of substitutions ranging in ground constructor terms.

2.2.3 Rank, precedence and call-trees

A *precedence* for a signature is a pre-order on symbols. Let $\prec$ denotes such a precedence, it induces an equivalence relation denoted by $\approx$. The *rank* of a symbol is its equivalence class for $\approx$.

A program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ induces a precedence over function symbols. Let $\prec$ be the transitive closure of :

$$g \quad \prec \quad f \text{ if } f(p_1, \ldots, p_n) \to r \in \mathcal{R} \text{ with } g(t_1, \ldots, t_n) \trianglelefteq r.$$

By default, when we speak about the rank of a symbol, without mentioning some particular precedence, it is the rank induced by the program rules.

Definition 11 (Call-tree)**.** Suppose we are given a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$. Let $\rightsquigarrow$ be the relation

$$\begin{aligned} &(f, t_1, \ldots, t_n) \rightsquigarrow (g, u_1, \ldots, u_m) \\ &\Leftrightarrow \\ &f(t_1, \ldots, t_n) \to C[g(w_1, \ldots, w_m)] \wedge \forall i : w_i \xrightarrow{!} u_i \end{aligned}$$

with f and g some defined symbols, and $t_1, \ldots, t_n, u_1, \ldots, u_m$ are constructor terms. Given a term $f(t_1, \ldots, t_n)$, the relation $\rightsquigarrow$ defines a tree whose root is $(f, t_1, \ldots, t_n)$ and η' is a daughter of η iff $\eta \rightsquigarrow \eta'$. The size of a call-tree is the number of nodes it contains.

2.2.4 Computation by rewriting

As noted above, rewriting can be used as a computational model. Since we will refer to complexity theory, we give here some basic facts concerning aspects of computations by rewriting.

Proposition 12. *Given a rewriting system $\langle \Sigma, \mathcal{R} \rangle$, there is a constant K such that for all $s \to t$, we have $||s|| \leq ||t|| + K$. There is also a constant K' such that for all $s \to t$, $|s| \leq K'(|t| + 1)$.*

Proposition 13. *Given a rewriting system $\langle \Sigma, \mathcal{R} \rangle$, one step of computation can be performed in time $O(n)$ where n is the size of the input.*

Performing one step of computation amounts to finding a redex —that is a position where a rule may apply— by some unification procedure, and then, to replace it by the corresponding right hand side. Both operations can be done in $O(n)$.

Actually, a lot of implementations of rewriting have been proposed, among wich we note the work around ELAN [136]. All these implementations fit into the framework of Proposition 12.

Finally, the work of dal Lago and Martini shows that the derivation height can be used safely to evaluate the complexity of rewriting [52]. An other reference is Avanzini and Moser [13]. We will use their result restated in the following way :

Corollary 14 (dal Lago, Martini)**.** *Let φ be computed by some F-program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$. Let us suppose that for all terms $f(t_1, \ldots, t_n)$ with $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C})$ both the derivation height and the size of the normal form is polynomially bounded (resp. exponentially, doubly exponentially). Then φ can be computed in polynomial time (resp. exponential time, doubly exponential time).*

2.3 Termination

We say that a rewriting system $\langle \Sigma, \mathcal{R} \rangle$ is *terminating* if the relation $\to$ is well founded, that is there is no infinite sequence $t_0 \to t_1 \to \cdots$.

Finding new methods to prove termination has been an active field of research in the last years. One of the pioneers is Dershowitz whose work initiated many methods of termination. We refer the reader to [55] for a more detailed description of this work.

2.3.1 Recursive Path ordering

Recursive Path Orderings are simplification orderings and so well-founded. Among the pioneers of this subject,let us cite Plaisted [126], Dershowitz [54], Kamin and Lévy [91]. Finally, Krishnamoorthy and Narendran in [94] have proved that deciding whether a program terminates by Recursive Path Orderings or not is a NP-complete problem.

Extension of an ordering to sequences Suppose that $\preceq$ is a partial ordering on some set A. Let $\prec$ denotes its strict part. We describe two extensions of $\prec$ to sequences of elements of A.

$$\frac{s \trianglelefteq t_i \ or \ s \prec_{rpo} t_i}{s \prec_{rpo} \mathtt{f}(\ldots, t_i, \ldots)} \ \mathtt{f} \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{\forall i \ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathbf{c}(s_1, \cdots, s_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \ \mathtt{f} \in \mathcal{F}, \mathbf{c} \in \mathcal{C}$$

$$\frac{\forall i \ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{g(s_1, \cdots, s_m) \prec_{rpo} \mathtt{g}(t_1, \cdots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

$$\frac{(s_1, \cdots, s_n) \prec_{rpo}^{st(\mathtt{f})} (t_1, \cdots, t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad \forall i \ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

FIGURE 2.2 – Definition of $\prec_{rpo}$

Definition 15. The multiset extension of $\prec$ over sequences, denoted $\preceq^m$, is defined as the reflexive transitive closure of :

$$\begin{aligned} (s_1, \ldots, s_n) &\preceq^m (s_{\pi(1)}, \ldots, s_{\pi(n)}) \text{ with } \pi \text{ a permutation} \\ (s_1, \ldots, s_n) &\preceq^m (t) \text{ if } s_i \prec t \text{ for all } i \leq n \\ s.s' &\preceq^m t.t' \text{if } s \preceq^m t \wedge s' \preceq^m t' \end{aligned}$$

where, in the last equation, the "." corresponds to the concatenation of sequences. In other words, one can replace in a sequence any element by finitely many smaller ones. $\prec^m$ is the strict part of $\preceq^m$.

The lexicographic extension of $\prec$, denoted $\prec^l$, is defined as follows.

Definition 16. $(m_1, \cdots, m_k) \prec^l (n_1, \cdots, n_k)$ if and only if there exists an index j such that (i) $\forall i < j, m_i \preceq n_i$ and (ii) $m_j \prec n_j$.

Recursive path ordering with status

Definition 17. A *status st* is a mapping which associates to each function symbol $\mathtt{f}$ of $\mathcal{F}$ an element in $\{m, l\}$. A status is compatible with a precedence $\preceq_{\mathcal{F}}$ if for all $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ then $st(\mathtt{f}) = st(\mathtt{g})$. Throughout, we assume that status are compatible with precedences.

Definition 18. Given a precedence $\preceq_{\mathcal{F}}$ and a compatible status st, the recursive path ordering $\prec_{rpo}$ is defined in Figure 2.2.

When $st(\mathtt{f}) = m$, the status of $\mathtt{f}$ is said to be multiset. In that case, the arguments are compared with the multiset extension of $\prec_{rpo}$. Otherwise, the status is said to be lexicographic.

A program is ordered by $\prec_{rpo}$ if there is a precedence on $\mathcal{F}$ and a status st such that for each rule $\ell \to r$, $r \prec_{rpo} \ell$. We say that a program is multiset if all function symbols have multiset status. Otherwise, we say that the program is lexicographic. The set of multiset (resp. lexicographic) programs is written $\mathtt{F}.mpo$ (resp. $\mathtt{F}.rpo$).

Theorem 19 (Dershowitz [54])**.** *Each program which is ordered by $\prec_{rpo}$ terminates on all inputs.*

Example 20.

1. The shuffle program rearranges two words. It terminates with a multiset status.

$$\begin{aligned}\text{shuffle}(\boldsymbol{\varepsilon}, y) &\to y \\ \text{shuffle}(x, \boldsymbol{\varepsilon}) &\to x \\ \text{shuffle}(\mathbf{i}(x), \mathbf{j}(y)) &\to \mathbf{i}(\mathbf{j}(\text{shuffle}(x, y))) \qquad \mathbf{i}, \mathbf{j} \in \{\mathbf{0}, \mathbf{1}\}\end{aligned}$$

2. The following program reverses a word by tail-recursion. It terminates with a lexicographic status. Since $y \lhd \mathbf{s}(y)$, it cannot be ordered with a multi-set status.

$$\begin{aligned}\text{reverse}(\boldsymbol{\varepsilon}, y) &\to y \\ \text{reverse}(\mathbf{i}(x), y) &\to \text{reverse}(x, \mathbf{i}(y)) \qquad \mathbf{i} \in \{\mathbf{0}, \mathbf{1}\}\end{aligned}$$

3. The program `sort` of Example 5 terminates by MPO. One sets the precedence to be `if then else` $\prec_{\mathcal{F}}$ `insert` $\prec_{\mathcal{F}}$ `sort`
4. The `lcs` program of Example 9 is ordered by MPO, taking `max` $\prec_{\mathcal{F}}$ `lcs`.

2.4 Interpretations of programs

Given a signature Σ, an *assignment* is a mapping $(\!| - |\!)$ which associates to every n-ary symbol $f \in \Sigma$ an n-ary function $(\!| f |\!) : A^n \to A$. Such a Σ-algebra can be extended to terms by :
- $(\!| x |\!) = 1_A$, that is the identity on A, for $x \in \mathcal{X}$,
- $(\!| f(t_1, \ldots, t_m) |\!) = \text{comp}((\!| f |\!), (\!| t_1 |\!), \ldots, (\!| t_m |\!))$ where comp is the composition of functions.

Given a term t with n variables, $(\!| t |\!)$ is a function $A^n \to A$.

Definition 21. Let $(A, \leq)$ be a given a partially ordered set and $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ be a program. As usual, $<$ denotes the strict ordering induced by $\leq$. Let us consider a $(\mathcal{C} \cup \mathcal{F})$-algebra $(\!| - |\!)$ on A. It is said to :
- **(SMon)** be strictly monotonic if for any symbol f, the function $(\!| f |\!)$ is a strictly monotonic function, that is $x_i > x'_i$ implies
$$(\!| f |\!)(x_1, \ldots, x_n) > (\!| f |\!)(x_1, \ldots, x'_i, \ldots, x_n),$$
- **(WMon)** be weakly monotonic if for any symbol f, the function $(\!| f |\!)$ is a weakly monotonic function, that is $x_i \geq x'_i$ implies
$$(\!| f |\!)(x_1, \ldots, x_n) \geq (\!| f |\!)(x_1, \ldots, x'_i, \ldots, x_n),$$
- **(WSub)** have the weak sub-term property if for any symbol f, the function $(\!| f |\!)$ verifies $(\!| f |\!)(x_1, \ldots, x_n) \geq x_i$ with $i \in 1, \ldots, n$,
- **(SCmp)** to be strictly compatible (with the rewriting relation) if for all rules $\ell \to r$, $(\!| \ell |\!) > (\!| r |\!)$ where $>$ denotes the pointwise ordering on functions.
- **(WCmp)** to be weakly compatible if for all rules $\ell \to r$, $(\!| \ell |\!) \geq (\!| r |\!)$,
- **(Cns)** to be conservative if for all rule $\ell \to r$ and all $u \trianglelefteq r$ there is a subterm $v \trianglelefteq \ell$ such that $(\!| u |\!) \leq (\!| v |\!)$,
- **(NFA)** to be a normal form size approximation if for all constructor terms $t_1, \ldots, t_n$, the inequality $(\!| f(t_1, \ldots, t_n) |\!) \geq (\!| [\![f]\!](t_1, \ldots, t_n) |\!)$ holds.

Definition 22. Given an ordered set $(A, \leq)$ and a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$, a $(\mathcal{C} \cup \mathcal{F})$-algebra on A is said to be :

- a strict interpretation whenever it verifies (SMon), (WSub), (SCmp),
- a quasi-interpretation whenever it verifies (WMon), (WSub), (WCmp),
- a conservative interpretation whenever it verifies (WMon), (Cns), (WCmp),
- a monotone interpretation whenever it verifies (WMon) and (WCmp),
- a sup-interpretation if it verifies (WMon), (WCmp).

Example 23. The program given in Example 8 has both a strict interpretation (left side, black) and a conservative interpretation (right side, blue) :

$$
\begin{array}{rl}
(\!|\mathbf{tt}|\!) = (\!|\mathbf{ff}|\!) & = 1 \\
(\!|\varepsilon|\!) = (\!|\mathtt{nil}|\!) & = 1 \\
(\!|\mathbf{i}|\!)(\mathbf{x}) & = x+1 \\
(\!|\mathbf{cons}|\!)(x,y) & = x+y+3 \\
(\!|\mathtt{not}|\!)(x) & = x+1 \\
(\!|\mathtt{or}|\!)(x,y) & = x+y+1 \\
(\!|\mathtt{and}|\!)(x,y) & = x+y+1 \\
(\!|=|\!)(x,y) & = x+y+1 \\
(\!|\mathtt{if\ then\ else}|\!)(x,y,z) & = x+y+z \\
(\!|\mathtt{in}|\!)(x,y) & = (x+1)(y+1)
\end{array}
\qquad
\begin{array}{rll}
(\!|\mathbf{tt}|\!) = (\!|\mathbf{ff}|\!) & = 1 & \\
(\!|\varepsilon|\!) = (\!|\mathtt{nil}|\!) & = 1 & \\
(\!|\mathbf{i}|\!)(\mathbf{x}) & = x+1 & \text{with } \mathbf{i} \in \{0,1\} \\
(\!|\mathbf{cons}|\!)(x,y) & = x+y+1 & \\
(\!|\mathtt{not}|\!)(x) & = 1 & \\
(\!|\mathtt{or}|\!)(x,y) & = 1 & \\
(\!|\mathtt{and}|\!)(x,y) & = 1 & \\
(\!|=|\!)(x,y) & = 1 & \\
(\!|\mathtt{if\ then\ else}|\!)(x,y,z) & = \max(y,z) & \\
(\!|\mathtt{in}|\!)(x,y) & = 1 &
\end{array}
$$

Strict interpretation were introduced in the seventies by Lankford [98] to prove termination of rewriting systems. In [98], $(A, \leq)$ is the set of natural numbers with their usual ordering.

Quasi-interpretations have been defined in [26] (called pseudo-interpretations) and in [107] . Sup-interpretation have been introduced by Marion and Pechoux in [109]. We gave here a slight variant of their definition. In [109], the last inequality refers to the size of normal forms. We prefered to use a more uniform definition.

Clearly, a strict interpretation is a quasi-interpretation that is itself a monotone interpretation which, finally, is a sup-interpretation. Actually, (WSub) and (WCmp) implies (Cns). As a consequence, a quasi-interpretation is a conservative interpretation. Observe also that a conservative interpretation is itself a monotone interpretation.

In the sequel, we will refer equally to one of these notions by the generic word "interpretation". We also use this terminology to speak about the function $(\!|f|\!)$ given a symbol f.

Remark 24. On natural numbers, and actually on any initial segment of the ordinals, (WSub) is a consequence of (SMon).

Example 25. Let us consider the one-rule rewriting system :

$$f(a) \to f(a).$$

It has the following quasi-interpretation :

$$
\begin{array}{rcl}
(\!|f|\!)(x) & = & x \\
(\!|a|\!) & = & 1
\end{array}
$$

with $(A, \leq)$ being the set of natural numbers with their usual ordering. Due to (SCmp), such a system has no strict interpretation.

Actually, every rewriting system admits a quasi-interpretation. Take $A = \mathbf{N}$ with its usual order and for all n-ary symbol $f \in \Sigma$, set $(\!| f |\!)(X_1, \ldots, X_n) = \max(X_1, \ldots, X_n)$. It is clear that the hypotheses of the definition are fulfilled. To discriminate low complexity (good) programs from the bad ones, it makes then sense to consider some further restrictions on quasi-interpretations. This will be discussed in more details in Section 3.

Proposition 26 (Compatibility with rewriting)**.** *Given a rewriting system $\langle \Sigma, \mathcal{R} \rangle$ and an interpretation $[\![-]\!]$, the following holds :*

1. *If (SMon) and (SCmp), then for all closed terms $t \xrightarrow{+} u$, $(\!| t |\!) > (\!| u |\!)$ holds.*
2. *If (WMon) and (WCmp), then for all closed terms $t \xrightarrow{*} u$, $(\!| t |\!) \geq (\!| u |\!)$ holds.*

The property (1) holds for strict interpretations, (2) holds for quasi-interpretations, conservative interpretations and monotone interpretations.

Proposition 27 (Sub-term property)**.** *If (WMon) and (WSub), then for all closed terms $t \trianglelefteq u$, the inequality $(\!| t |\!) \leq (\!| u |\!)$ holds.*

The property holds for strict-interpretations and quasi-interpretations.

Proposition 28 (Conservativity)**.** *If (WMon), (WCns) and (WCmp), for all closed terms $t \xrightarrow{*} u$ and all $v \trianglelefteq u$, there is some $w \trianglelefteq t$ such that $(\!| v |\!) \leq (\!| w |\!)$.*

The property holds for strict-interpretation, quasi-interpretations and conservative interpretations.

To conclude, let us summarize the differences between these notions of interpretations :

property	strict-interpretation	quasi-interpretation	conservative interpretation	monotone-interpretation	sup-interpretation
$s \trianglelefteq t$	$(\!| s |\!) \leq (\!| t |\!)$	$(\!| s |\!) \leq (\!| t |\!)$			
$\ell \to r$	$(\!| \ell |\!) > (\!| r |\!)$	$(\!| \ell |\!) \geq (\!| r |\!)$	$(\!| \ell |\!) \geq (\!| r |\!)$	$(\!| \ell |\!) \geq (\!| r |\!)$	
$t \xrightarrow{+} u$	$(\!| t |\!) > (\!| u |\!)$	$(\!| t |\!) \geq (\!| u |\!)$	$(\!| t |\!) \geq (\!| u |\!)$	$(\!| t |\!) \geq (\!| u |\!)$	
$t \xrightarrow{*} u \trianglerighteq v$	$(\!| v |\!) \leq (\!| t |\!)$	$(\!| v |\!) \leq (\!| t |\!)$	$\exists w \trianglelefteq t :$ $(\!| v |\!) \leq (\!| w |\!)$		
$t \xrightarrow{!} u$	$(\!| t |\!) > (\!| u |\!)$	$(\!| t |\!) \geq (\!| u |\!)$	$(\!| t |\!) \geq (\!| u |\!)$	$(\!| t |\!) \geq (\!| u |\!)$	$(\!| t |\!) \geq (\!| u |\!)$

Chapitre 3

First results in Implicit computational complexity

This chapter is devoted to the characterization of functions computed by programs with an interpretation over the naturals. We mainly focus on complexity classes. Some of the results stated in this section follow from the articles [30] and [38].

It is clear that the choice of the ordering for the interpretation plays a crucial role for the characterization of computed functions. An other key feature is the set of functions in which interpretations may vary. Actually there is a kind of tradeoff between the complexity of the ordering (corresponding to high ordinals) and the complexity of the interpretations (corresponding to high growth rate). Let us illustrate this observation on a simple example.

Consider the Ackermann function [5]. It is computed by the rewrite system :

$$\begin{array}{rcl} A(0,n) & \rightarrow & \mathbf{s}(n) \\ A(\mathbf{s}(m),0) & \rightarrow & A(m,\mathbf{s}(0)) \\ A(\mathbf{s}(m),\mathbf{s}(n)) & \rightarrow & A(m,A(\mathbf{s}(m),n)) \end{array}$$

To prove its termination, we introduce an ordering on finite trees due to Jervell [86]. Let us consider the set T of finite rooted trees. The one-node tree is written ".", otherwise, trees are denoted by $\mathbf{A} = \overset{x_1 \cdots x_n}{\diagdown\,\diagup}$. The sequence $x_1, \ldots, x_n$ of $\mathbf{A}$ is denoted by $\langle \mathbf{A} \rangle$. Given an ordering $\prec$ on trees, we define

- $\mathbf{A} \preceq \langle \mathbf{B} \rangle$ iff $\exists i : \mathbf{A} \prec \mathbf{B}_i$ or $\mathbf{A} = \mathbf{B}_i$,
- $\langle \mathbf{A} \rangle \prec \mathbf{B}$ iff $\forall j : \mathbf{A}_j \prec \mathbf{B}$,
- $\langle \mathbf{A} \rangle \prec \langle \mathbf{B} \rangle$ iff
  - if $|\mathbf{A}| < |\mathbf{B}|$, that is $\langle \mathbf{A} \rangle$ is a shorter sequence than $\langle \mathbf{B} \rangle$,
  - or it is ordered according to the inverse (from right to left) lexicographic ordering.

We define $<$ on T to be the smallest ordering such that $\mathbf{A} < \mathbf{B} \Leftrightarrow \mathbf{A} \leq \langle \mathbf{B} \rangle \vee (\langle \mathbf{A} \rangle < \mathbf{B} \wedge \langle \mathbf{A} \rangle < \langle \mathbf{B} \rangle)$. This relation is a linear, well founded order, that is, it corresponds to an initial segment of the ordinals. Actually, when restricted to binary trees, it is ε_0 (cf. [86]).

5. We use a variant of the function due to Peter [125].

The following equations define an interpretation for the Ackermann's function :

$$
\begin{array}{rcl}
(\!| 0 |\!) & = & \cdot \\
(\!| S |\!)(x) & = & \begin{array}{c} x \\ \vdots \end{array} \\
(\!| A |\!)(x, y) & = & \begin{array}{c} y \;\; x \\ \setminus\!/ \end{array}
\end{array}
$$

One may observe that the size of the output of these functions is polynomial w.r.t. the size of their inputs (actually, it is linear). In that sense, the complexity of the computation is encoded in the ordering, not in the complexity of the function used for the interpretation. A contrario, when we use a small ordering as a universe for interpretations, we need some functions with high growth rate.

The following is an interpretation over integers.

$$
\begin{array}{rcl}
(\!| 0 |\!) & = & 2 \\
(\!| S |\!)(x) & = & x + 2 \\
(\!| A |\!)(x, y) & = & A(x, y)
\end{array}
$$

Indeed, we have for the first rule :

$$
\begin{array}{rcl}
(\!| A(0, n) |\!) & = & A(2, n) \\
& = & 2n + 3 \\
& > & n + 2 = (\!| \mathbf{s}(n) |\!)
\end{array}
$$

For the second rule :

$$
\begin{array}{rcl}
(\!| A(\mathbf{s}(n), 0) |\!) & = & A(n + 2, 2) \\
& > & A(n, 4) = (\!| A(n, \mathbf{s}(0)) |\!) \text{ See the footnote}^{6}
\end{array}
$$

And for the last rule :

$$
\begin{array}{rcl}
(\!| A(\mathbf{s}(n), \mathbf{s}(m)) |\!) & = & A(n + 2, m + 2) \\
& = & A(n + 1, A(n + 2, m + 1)) \\
& > & A(n, A(n + 2, m)) \text{ by monotonicity} \\
& > & (\!| A(n, A(\mathbf{s}(n), m)) |\!)
\end{array}
$$

---

6. Proved by case. For $n \leq 1$, $A(2, 2) = 7 > 5 = A(0, 4)$ and $A(3, 2) = 17 > 6 = A(1, 4)$. Otherwise, for $n > 1$, noting that $A(n + 1, m) \geq A(n, m) + 1$ and $A(n, m + 1) \geq A(n - 1, A(n, m)) \geq A(1, A(n, m)) > A(n, m) + 1$, we have :

$$
\begin{array}{rcl}
A(n + 2, 2) & = & A(n + 1, A(n + 1, A(n, 1))) \\
& \geq & A(n + 1, A(n, A(n, 1) + 1)) \text{ see remark above} \\
& > & A(n + 1, A(n, A(n, 1)) + 1) \text{ see remark above} \\
& > & A(n, A(n, A(n, A(n, 1)))) \text{ expanding the definition for the first "A"} \\
& > & A(n, A(n, A(n, A(n, 1)))) = A(n, 4) \text{ monotonicity}
\end{array}
$$

Actually, it is not possible to prove the termination of Ackermann's function if one restricts assignments to be primitive recursive. Ad absurdum, suppose that we have an interpretation for Ackermann's rewrite system with primitive recursive assignments. Consequently, the function $n \mapsto (\!|\mathtt{ack}(\mathbf{s}^n(\mathbf{0}), \mathbf{0})|\!)$ is primitive recursive. Indeed, the function $n \mapsto (\!|\mathbf{s}^n(\mathbf{0})|\!)$ is primitive recursive and the function $n \mapsto (\!|\mathtt{ack}(\mathbf{s}^n(\mathbf{0}), \mathbf{0})|\!) = (\!|\mathtt{ack}|\!)((\!|\mathbf{s}^n(\mathbf{0})|\!), (\!|\mathbf{0}|\!))$ is primitive recursive by composition.

By Proposition 29, the derivation height of the system should be primitive recursive, which is not the case.

Similar analysis are carried out in the field of sub-recursive functions where people have developed some tools to classify sets of (computable) functions by means of ordinal notations [132, 58]. The higher ordinals are, the larger are the sets of functions embraced by the theory. One of the essential features of these constructions is to provide, for each set S of functions, one function (a *snake* in Simmons's terminology) which is a bound [7] of any function in S.

The main difference between sub-recursive analysis and analysis by interpretations comes from the following point. It is well known that snakes can serve both as a bound of the computation length and as a bound of the size of the output, see for instance [128]. An interpretation gives a bound on the sum of the computation length and the size of the output.

In what follows, we focus on functions with feasible complexity, say below exponential time. High ordinals provide large sets of functions, but also, comme un mal nécessaire, functions with high complexity. So, from now on, we will not use the order as a parameter of our analysis, restricting our attention to interpretations over the integers, or reals with their usual order.

3.1 Interpretation over N

Using natural numbers with their usual order as a universe of interpretation conveys some particular properties that we present now. Actually, compared to what happens with real numbers, the use of natural numbers make the proofs direct and much more easy. For that reason, the study of interpretations over the reals is postponed.

The set of programs with a strict interpretation (resp. quasi-interpretation, sup-interpretation) is written $\mathtt{F}.SI$ (resp. $\mathtt{F}.QI, \mathtt{F}.SPI$)

First, observe that the derivation height of a term t can be immediately bounded by its interpretation.

Proposition 29. *Let (Σ, R) be a rewriting system and $(\!|-|\!)$ be an interpretation over the natural numbers. Then, for all $t \in \mathcal{T}(\Sigma)$, we have*

$$\mathtt{dh}(t) \leq (\!|t|\!).$$

Since the interpretations of terms depend essentially on the growth rate of the functions used for the assignment, so is the derivational complexity of the system, and consequently the complexity of computed functions. To capture programs with a small complexity, it is then reasonable to consider some set of functions with small growth rate.

3.1.1 A tiering of interpretations

Definition 30 (Polynomially bounded interpretation)**.** We say that an interpretation is polynomially bounded if for all symbol $f \in \Sigma$, $(\!|f|\!)$ is bounded by a polynomial. That is for all f, there is a polynomial P_f such that $(\!|f|\!)(\vec{x}) \leq P_f(\vec{x})$.

7. up to composition

Definition 31.

- An assignment of $f \in \Sigma$ is *additive* if

$$(\!| f |\!)(x_1, \cdots, x_n) = \sum_{i=1}^{n} x_i + \alpha \qquad \alpha > 0$$

- An assignment of f is *affine* if

$$(\!| f |\!)(x_1, \cdots, x_n) = \sum_{i=1}^{n} \beta_i x_i + \alpha \qquad \alpha > 0$$

- Otherwise, an assignment of f is said to be *multiplicative* whenever $(\!| f |\!)(x_1, \ldots, x_n) > \sum_{i=1}^{n} x_i$.

Given a F-program with a polynomially bounded interpretation, we say that it is additive (resp. affine, multiplicative) if the assignment of its *constructors* is additive (resp. affine, multiplicative). We add the subscript $+$ (resp. λ, $\times$) to denote the set of additive (resp. affine, multiplicative) programs. Finally, we denote by **Max-Poly** the set of functions obtained by finite compositions of $\max, +, \times$ and constant functions. **Max-Plus** denotes the set of finite compositions of max and $+$ and constant functions.

Proposition 32. *Let $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ be an additive, affine or a multiplicative program. The following holds :*

1. *for all constructors* $\mathbf{c}$, $(\!| \mathbf{c} |\!)(x_1, \ldots, x_n) \geq \sum_{i=1}^{n} x_i + 1$,
2. *for all constructor terms* t, $|t| \leq (\!| t |\!)$,
3. *if the program is additive, then there is a constant* $K > 0$ *such that* $(\!| t |\!) \leq K \times |t|$ *for all* $t \in \mathcal{T}(\mathcal{C})$,
4. *if the program is affine (resp. multiplicative), then, there is a constant* K *such that* $(\!| t |\!) \leq 2^{K|t|} + K$ *(resp.* $(\!| t |\!) \leq 2^{2^{K|t|}} + K$*) for all* $t \in \mathcal{T}(\mathcal{C})$.

Démonstration. (1) is an immediate consequence of the definitions. (2–4) are proved by induction on t. □

Corollary 33. *For any additive, affine or multiplicative* F*-program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$, suppose that $u_1, \ldots, u_n \in \mathcal{T}(\mathcal{C})$. If $t[u_1, \ldots, u_n] \xrightarrow{*} v$ with $v \in \mathcal{T}(\mathcal{C})$, then $|v| \leq (\!| t(u_1, \ldots, u_n) |\!)$. So, the size of the outputs of programs is directly linked to the value of the initial interpretation. For additive programs, Proposition 32 leads to $|v| \leq (\!| t |\!)(K|u_1|, \ldots, K|u_n|)$. Consequently, the output of additive program has a polynomial size w.r.t. to its corresponding input.*

Example 34. Polynomials can be computed by composing addition and multiplication.

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

On the left, we provide a strict interpretation and on the right a quasi-interpretation.

$$\begin{aligned} (\!|\mathbf{0}|\!) &= 1 & (\!|\mathbf{0}|\!) &= 0 \\ (\!|\mathtt{s}|\!)(x) &= x+2 & (\!|\mathtt{s}|\!)(x) &= x+1 \\ (\!|\mathtt{add}|\!)(x,y) &= 2x+y & (\!|\mathtt{add}|\!)(x,y) &= x+y \\ (\!|\mathtt{mult}|\!)(x,y) &= (x+1).(y+1) & (\!|\mathtt{mult}|\!)(x,y) &= x \times y \end{aligned}$$

So, addition and multiplication are additive programs, both for strict-interpretations and quasi-interpretations. Now, in order to define the exponential, we introduce another successor $\mathbf{s}'$ which has an affine assignment.

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

On the left, we give an interpretation, on the right a quasi-interpretation.

$$\begin{aligned} (\!|\mathtt{s}'|\!)(x) &= 3x+10 & (\!|\mathtt{s}'|\!)(x) &= 2x+1 \\ (\!|\mathtt{exp}|\!)(x) &= x+3 & (\!|\mathtt{exp}|\!)(x) &= x+1 \end{aligned}$$

One will notice that we implicitly used the in-out semantics. Indeed, we see that the domain of `exp` and its co-domain are not the same. The domain is generated by $\{\mathbf{0}, \mathbf{s}'\}$ whose interpretation is affine and the co-domain is generated by $\{\mathbf{0}, \mathbf{s}\}$ whose interpretation is additive. Actually, even if one allows affine or multiplicative interpretations for constructors, there is no program to compute the exponential whenever the input domain and the output domains are equal (see [26], chap 3.).

We define the doubly-exponential function, i.e. $n \mapsto 2^{2^n}$, as follows.

$$\begin{aligned} \mathtt{dexp}(\mathbf{0}) &\to \mathtt{s}(\mathtt{s}(\mathbf{0})) \\ \mathtt{dexp}(\mathtt{s}''(x)) &\to \mathtt{mult}(\mathtt{dexp}(x), \mathtt{dexp}(x)) \end{aligned}$$

and

$$\begin{aligned} (\!|\mathtt{s}''|\!)(x) &= (x+6)^2+1 & (\!|\mathtt{s}''|\!)(x) &= (x+2)^2 \\ (\!|\mathtt{dexp}|\!)(x) &= x+5 & (\!|\mathtt{dexp}|\!)(x) &= x+2 \end{aligned}$$

Again we see that the domain and co-domain are not the same. The domain admits a multiplicative quasi-interpretation and the co-domain has an additive one.

3.2 Polynomial interpretations

In this section, we suppose that interpretations are chosen in **Max-Poly**. In that case, the following holds :

Theorem 35. *[after BCMT'01] Functions computed by F-programs with an additive interpretation are exactly* PTIME *functions. For higher tiers, we get respectively* ETIME *for affine programs and* E$_2$TIME *for multiplicative programs.*

Compared to [30], there are two differences, a) we only use the large sub-term property, not the strict one and b) we allow the use of the function max. Consequently, we loose the fact that $|t| \leq (\!|t|\!)$. This latter property ensures that for all computation $f(t_1,\ldots,t_n)\xrightarrow{*}t\xrightarrow{!}v$, $|t| \leq (\!|t|\!) \leq (\!|f(t_1,\ldots,t_n)|\!)$. In the present terms, the size of intermediate values cannot be bounded by the interpretations. Nevertheless, due to Proposition 29, the derivation length of additive programs remains polynomial : $\mathtt{dh}(f(t_1,\ldots,t_n)) \leq (\!|f|\!)((\!|t_1|\!),\ldots,(\!|t_n|\!)) \leq (\!|f|\!)(K|t_1|,\ldots,K|t_n|)$ for some $K > 0$. So, the main difficulty is about space, not time. However, from Proposition 32, we know that the size of the normal form is polynomial w.r.t. the size of the input. Then, from Corollary 14, the function can be computed in polynomial time.

3.3 Sup-interpretation

The theorems given in this section extend the ones given in [38]. There are two main differences, a) we use sup-interpretation, not quasi-interpretations and b) we employ the multiset path ordering, not the product path ordering (PPO) as it was done in previous work (e.g. [38]). In [111], Marion and Péchoux also use the product path ordering. Furthermore, they consider some further syntactic constraints on recursive calls that we skip.

For a), we can find a sup-interpretation for some rewriting system for which the subterm property is an issue. For instance, using the rule

$$\mathrm{sqrt}(x,y) \quad \to \mathtt{if}\ \ y\times y > x\ \mathtt{then}\ \ y-1\ \mathtt{else}\ \ \ \mathrm{sqrt}(x,\mathbf{s}(y))$$

$\mathrm{sqrt}(x,0)$ computes the square root of x. It has a sup-interpretation $(\!|\mathrm{sqrt}|\!)(x,y) = x$.[8]

For b), the product path ordering (PPO) cannot prove the termination of rules such as :

$$f(\mathbf{s}(x),y) \to f(x,x)$$

which is in the scope of the multi-set path ordering. More generally, the product path ordering does not cope with the duplications nor the erasing of variables.

Finally, with the product path ordering, for all constructor terms, $t \prec_{rpo}^{product} u$ iff $t \lhd u$. As a consequence, a rule such as :

$$f(1(0(x))) \to f(1(x))$$

is not compatible with PPO. The main issue with MPO is that the set of terms $U_t = \{u \mid u \prec_{rpo}^{m} t\}$ does not have a polynomial size w.r.t. the size of t, a key feature used in [38, 111] to apply memoisation. Indeed, one may observe that the sequences booleans $\{0,1\}^n \subset U_{\underbrace{01\cdots01}_{2n\text{ times}}}$. Thus, $|U_{\underbrace{01\cdots01}_{2n\text{ times}}}| \geq 2^n$.

Theorem 36. *Functions computed by* F*-programs with an additive sup-interpretation and a multiset path ordering termination proof are exactly* Ptime *functions. For higher tiers, we get respectively* Etime *for affine programs and* E$_2$time *for multiplicative programs.*

The proof roughly follows the lines of the one given in [38]. Let us first recall some properties of the recursive path orderings. In the remaining of the section, we suppose given a F-program which is ordered by $\prec_{rpo}^{m}$, together with an additive sup-interpretation $(\!|\,|\!)$.

Proposition 37.

8. $(\!|-|\!)(x,y) = x$, $(\!|\mathtt{if\ then\ else}|\!)(x,y,z) = \min(y,z)$, $(\!|\times|\!)(x,y) = x\times y$.

1. *For each constructor terms t and s, $s \prec_{rpo} t$ iff $s \blacktriangleleft t$.*
2. *For each constructor term $s_1, \cdots, s_n$ and $t_1, \cdots, t_m$, $(s_1, \cdots, s_n) \prec^x_{rpo} (t_1, \cdots, t_m)$ implies $(s_1, \cdots, s_n) \blacktriangleleft^x (t_1, \cdots, t_m)$, where x is a status m or l and $\blacktriangleleft^x$ is the corresponding extension based on the embedding relation.*
3. *If t is a constructor term and u contains a function symbol, then $u \not\prec^m_{rpo} t$.*
4. *If $(t_1, \cdots, t_n) \prec^m_{rpo} (u_1, \cdots, u_k)$, then, for all t_i, there is u_j such that $t_i \preceq^m_{rpo} u_j$.*

Démonstration. The proof is by induction on terms. □

Proposition 38. *Let us suppose given a program in F.mpo. For all rules $f(p_1, \cdots, p_n) \to r$, there is a context C such that*
- *$r = C[f_1(\vec{u_1}), \ldots, f_k(\vec{u_k})]$,*
- *C does not contain any symbol of rank of f,*
- *$f_1, \ldots, f_k$ have the rank of f,*
- *the $\vec{u_i}$'s are constructor terms.*

Démonstration. Let C be the largest context such that $r = C[t_1, \ldots, t_n]$ and
- C does not contain any symbol of rank of f
- for all i, $t_i = f_i(\vec{u_i})$ with f_i of rank of f.

The key point is that the patterns p_j are constructor terms. Then, for all $i \leq n$, since $t_i = f_i(\vec{u_i}) \preceq^m_{rpo} r \prec^m_{rpo} f(p_1, \cdots, p_n)$, it is necessarily the case that $(\vec{u_i}) \prec^m_{rpo} (p_1, \cdots, p_n)$. From 37 (3-4), the u_i's contain no function symbols. □

Proposition 39. *Let us suppose given a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ ordered by MPO, there is a finite set Θ such that :*
- *for all symbol $f \in \mathcal{F}$,*
- *for all constructor terms $t_1, \ldots, t_n$,*
- *for all constructor terms $u_1, \ldots, u_k$,*
- *for all function symbol g of rank of f such that $f(t_1, \ldots, t_n) \xrightarrow{*} C[g(u_1, \ldots, u_k)]$,*
- *for all $i \leq k$,*

there is an element $e \in \Theta$ and some positions $q_1, \ldots, q_m$ in $t_1, \ldots, t_n$ such that $u_i = e[t_{i_1|q_1}, \ldots, t_{i_m|q_m}]$.

Démonstration. Let Θ be the following set.
- $x \in \Theta$,
- for all $f(p_1, \cdots, p_n) \to r$, for all $i \leq n$, $p_i \in \Theta$,
- for all $p \trianglelefteq q, r \in \Theta$, $\text{ext}(p, q, r) \in \Theta$.

This set is finite. Indeed, $|\text{ext}(p, q, r)| \leq \max(|p|, |r|)$, so that $\Theta \subseteq \{v \mid |v| \leq d\}$ with $d = \max\{|p_i| \mid f(p_1, \cdots, p_n) \to r \in \mathcal{R}\}$.

Now, we proceed by induction on the length of the derivation.

For the base case, $f(t_1, \cdots, t_n) \xrightarrow{*} f(t_1, \cdots, t_n)$, the result is trivial (take $e = x, q = \varepsilon$).

Otherwise, $f(t_1, \cdots, t_n) \xrightarrow{*} C[g(u_1, \cdots, u_m)] \to C[C'[h(v_1, \cdots, v_k)]]$ after application of the rule $g(p_1, \cdots, p_m) \to C'[h(s_1, \cdots, s_k)]$ with the substitution σ. By Proposition 37, $s_i \trianglelefteq p_j$ for some j and $v_i = s_i[\sigma(x_{j_1}), \ldots, \sigma(x_{j_\ell})]$ for a subset $x_1, \ldots, x_{\ell'}$ of the variables occuring in p_j. By induction, $u_j = p_j[\sigma(x_1), \ldots, \sigma(x_{\ell'})] = e[t_{i_1|q_1}, \ldots, t_{i_m|q_m}]$ for some $e \in \Theta$ and some positions $q_1, \ldots, q_m$ in $t_1, \ldots, t_n$. Notice that $p_j \in \Theta$, since $s_i \trianglelefteq p_j$, $\text{ext}(s_i, p_j, e) \in \Theta$. Moreover, $v_i = \text{ext}(s_i, p_j, e)[t_{k_1|q_{m_1}.r_1}, \ldots, t_{k_h|q_{m_h}.r_h}]$ according to Proposition 2.

□

(Constructor)

$$\frac{\mathbf{c} \in \mathcal{C} \quad \langle C_{i-1}, t_i\rangle \Downarrow_c \langle C_i, v_i\rangle}{\langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \Downarrow_c \langle C_n, \mathbf{c}(v_1, \cdots, v_n)\rangle}$$

(Read)

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow_c \langle C_i, v_i\rangle \quad (\mathtt{f}, v_1, \cdots, v_n, v) \in C_n}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow_c \langle C_n, v\rangle}$$

(Update)

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow_c \langle C_i, v_i\rangle \quad \mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \langle C_n, r\sigma\rangle \Downarrow_c \langle C, v\rangle}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow_c \langle C \cup (\mathtt{f}, v_1, \cdots, v_n, v), v\rangle}$$

FIGURE 3.1 – Call-by-value interpreter with Cache of $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R}\rangle$.

Corollary 40. *Given a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R}\rangle$ ordered by* MPO*, $f \in \mathcal{F}$ and constructor terms $t_1, \ldots, t_n$, we denote by $T_{(f,t_1,\ldots,t_n)}$ the call tree rooted by $(f, t_1, \ldots, t_n)$ and by $R_{(f,t_1,\ldots,t_n)}$ the subset of nodes $(g, u_1, \ldots, u_m) \in T_{(f,t_1,\ldots,t_n)}$ such that $g \simeq f$. There is a polynomial P such that $|R_{(f,t_1,\ldots,t_n)}| \leq P(\max(|t_1|, \ldots, |t_n|))$.*

Démonstration. As seen above, if $(g, u_1, \ldots, u_m) \in R_{(f,t_1,\ldots,t_n)}$, then for all $i \leq m$, $u_i = e[t_{i_1|q_1}, \ldots, t_{i_m|q_m}]$ for some $e \in \Theta$ as defined above. One may observe that the number of positions for each q_i is bounded by $\sum_i |t_i|$. Let d be the maximal arity of the contexts $e \in \Theta$. Without loss of generality, we suppose d to be greater than the arity of any symbols in $\mathcal{F}$. We conclude, taking $P(x) = |\mathcal{F}| \times |\Theta| \times (d \times x)^{2d}$. □

Lemma 41. *Let $\mathtt{f}$ be a functional program with an additive sup-interpretation. Then, there is a polynomial $Q_{\mathtt{f}}$ such that for all equation $\ell \to r$ and all $e \trianglelefteq r$, $|[\![e]\!](t_1, \ldots, t_n)| \leq Q(\max(|t_1|, \ldots, |t_n|))$ with $t_1, \ldots, t_n$ constructor terms.*

Démonstration. It is proved by induction on the structure of e, using Corollary 33. □

Theorem 36. In order to avoid an exponential number of function calls like, for instance, in the `lcs` case, we switch from the call-by-value semantics previously defined to a call-by-value semantics with cache, see Figure 3.1. In other words, we simulate a dynamic programming technique inspired from Andersen and Jones' rereading ([8]) of Cook simulation technique over 2 way push-down automata ([49]) also known as memoization.

Our general purpose is to show that the size of the cache remains polynomial w.r.t. the size of the input, that is

- there is a polynomial number of entries and
- each entry has polynomial size, in other words :
  - the size of the arguments of calls are polynomial
  - the size of output terms is polynomial

Since each step of the call-by-value semantics with cache can be performed in polynomial (actually linear) time w.r.t. the size of the cache, it suffices to ensure that the computation can be done in polynomial time.

Actually, one must also note that functions computed with additive program have output of polynomial size w.r.t. their input (Corollary 33). Consequently, to ensure that the cache has

polynomial size, one only need to verify that arguments of function calls in the cache have polynomial size.

We will proceed by induction on the rank of symbols, and consequently, we provide a polynomial P_f to bound the size of the cache of the function f, depending only on the rank of f. Without loss of generality, we can suppose that $P_f \geq P_g$ if $g \prec f$.

First, let us define D to be the maximal arity of functions, A to be the number of function symbols and R to be the maximal size of the right-hand side of a rule. Finally, let us suppose we want to compute $f(t_1, \ldots, t_n)$. Elements in the cache will be terms $g(u_1, \ldots, u_m)$ such that $f(t_1, \ldots, t_n) \xrightarrow{*} C[g(u_1, \ldots, u_m)]$. Consequently, g has rank smaller or equal to the rank of f.

Base case Consider the evaluation of functions with minimal rank. Let us consider such a function f. Suppose that we are given a term $f(t_1, \ldots, t_n)$. Consider a term $g(u_1, \ldots, u_m)$ in the cache, since g has rank equivalent to the one of f, by Corollary 40, there are at most $P_D(\max(|t_1|, \ldots, |t_n|))$ such nodes. Due to Proposition 37(1,4), each of these argument has size bounded by $\max(|t_1|, \ldots, |t_n|)$. This defines the polynomial P_f.

Induction step Now, suppose f being of a higher rank. Suppose we want to compute $f(t_1, \ldots, t_n)$. Consider its call-tree $T_{f(t_1,\ldots,t_n)}$. There is actually a bijection between the set $S_{(f,t_1,\ldots,t_n)} = \{(g, v_1, \ldots, v_n) \mid (g, v_1, \ldots, v_n) \in T_{f(t_1,\ldots,t_n)}\}$ and the set of entries in the cache $\mathrm{Cache}_{f(t_1,\ldots,t_n)}$. Each element $(g, v_1, \ldots, v_k, v) \in \mathrm{Cache}_{f(t_1,\ldots,t_n)}$ corresponds to some nodes $(g, v_1, \ldots, v_k) \in T_{f(t_1,\ldots,t_n)}$. To sum up, it is sufficient to count the number of elements in $S_{(f,t_1,\ldots,t_n)}$ verifying that their arguments have polynomial size.

Suppose that $f(t_1, \ldots, t_n) \xrightarrow{*} C[g(u_1, \ldots, u_m)]$ with g of equal rank to the one of f. Again, by Proposition 40, there are only $P(\max(|t_1|, \ldots, |t_n|))$ entries in the cache for some polynomial P. Each of these arguments having size smaller than $\max(|t_1|, \ldots, |t_n|)$ by embedding.

So, the main issue is to evaluate the number of entries in the cache involving functions of smaller rank. Let $\rightsquigarrow_f$ be the following binary relation. $(g, v_1, \ldots, v_k) \rightsquigarrow_f (h, w_1, \ldots, w_m)$ holds iff $(g, v_1, \ldots, v_k) \rightsquigarrow^+ (h, w_1, \ldots, w_\ell) \wedge h \prec f$. Let $D_{(g,u_1,\ldots,u_k)} = \{(h, w_1, \ldots, w_\ell) \mid (g, v_1, \ldots, v_k) \rightsquigarrow_f (h, w_1, \ldots, w_\ell)\}$. The set of nodes in $S_{(f,t_1,\ldots,t_n)}$ is :

$$S_{(f,t_1,\ldots,t_n)} = R_{(f,t_1,\ldots,t_n)} \cup \bigcup_{(g,u_1,\ldots,u_m) \in R_{(f,t_1,\ldots,t_n)}} D_{(g,u_1,\ldots,u_m)}. \tag{3.1}$$

Let us compute the size of $D_{(g,u_1,\ldots,u_m)}$ for some node $(g, u_1, \ldots, u_m) \in R_{(f,t_1,\ldots,t_n)}$. The relation $\rightsquigarrow_f$ can be reformulated as $(g, u_1, \ldots, u_m) \rightsquigarrow_f (h, w_1, \ldots, w_k)$ iff

$$(g, u_1, \ldots, u_m) \rightsquigarrow h(w_1, \ldots, w_k) \wedge h \prec f \tag{3.2}$$

$$\text{or } (g, u_1, \ldots, u_m) \rightsquigarrow h'(u'_1, \ldots, u'_{k'}) \rightsquigarrow_f (h, w_1, \ldots, w_k) \text{ with } h' \prec f \tag{3.3}$$

Let $G_{(g,u_1,\ldots,u_m)}$ denotes the set of nodes verifying Equation (3.2). Let us consider a node $(h, w_1, \ldots, w_k) \in G_{(g,u_1,\ldots,u_m)}$. Suppose that there is a polynomial Q such that $|w_i| \leq Q(\max(|t_1|, \ldots, |t_n|))$ for all $i \leq k$. Then, by induction, the cache corresponding to the computation of $h(w_1, \ldots, w_k)$ is bounded by $P_h(Q(\max(|t_1|, \ldots, |t_n|)))$. Consequently, the size of $D_{(g,u_1,\ldots,u_m)}$ is bounded by $\sum_{(h,w_1,\ldots,w_k) \in G_{(g,u_1,\ldots,u_m)}} P_h(Q(\max(|t_1|, \ldots, |t_n|)))$. Recall that $|G_{(g,u_1,\ldots,u_m)}| \leq R$. Consequently, taking P_H a polynomial bounding P_h for all such h, we get the bound $|D_{(g,u_1,\ldots,u_m)}| \leq R \times P_H(Q(\max(|t_1|, \ldots, |t_n|)))$.

Recalling Equation (3.1), we can state that the cache of $f(t_1, \ldots, t_n)$ is bounded by

$$P(\max(|t_1|, \ldots, |t_n|)) \times (1 + R \times P_H(Q(\max(|t_1|, \ldots, |t_n|))))$$

which is a polynomial in $\max(|t_1|, \ldots, |t_n|)$, thus completing the induction.

So, it remains to prove the existence of Q as defined above. Recalling the definition of $\rightsquigarrow$, for all $(h, w_1, \ldots, w_k) \in G_{(g,u_1,\ldots,u_m)}$, we have $g(u_1, \ldots, u_m) \to r\sigma = C[h(e_1, \ldots, e_k)]\sigma$ for some context C and some terms e_i with $e_i\sigma \overset{!}{\to} w_i$. Observe that terms in the range of σ have size bounded by $\max(|u_1| \ldots, |u_m|)$, themselves bounded by $\max(|t_1|, \ldots, |t_n|)$. So, employing Lemma 41, we can state that $e_i\sigma$ has size polynomially bounded by $Q_{e_i}(\max(|t_1|, \ldots, |t_n|))$. Since there are only finitely many (there are finitely many rules) such e_i, the proof is done. □

Theorem 42 (B.-Marion-Moyen'07)**.** *Functions computed by* **F***-programs with an additive quasi-interpretation and a lexicographic ordering are exactly* PSPACE *functions. For higher tiers, we get respectively* ETIME *for affine programs and* E_2TIME *for multiplicative programs.*

This result can be related to the work of Leivant and Marion's about recurrence with parameter substitutions [102] : the lexicographic ordering endows such recurrence pattern. Theorem 36 holds both for quasi-interpretations and sup-interpretations. For the lexicographic path ordering, we need a stronger condition on the interpretations. Indeed, the Theorem holds :

Theorem 43. *Functions computed by* **F***-programs with an additive monotone interpretation and a lexicographic ordering are exactly functions computable by primitive recursion with polynomial output.*

Démonstration. Due to the encoding of sequences in the third layer of Grzegorczyk (see for instance Rose [128]), it is well known that that multiply recursive functions bounded by a polynomial output are actually primitive recursive, and even more, in the third layer of Grzegorczyk Hierarchy. So, the main point is to show that this class is complete. We prove that we can compute the function E_4, that is tower of exponentials. With such a clock bounding all functions in E_3, it is then routine to simulate a Turing Machine working in E_3 and, from that, obtaining the characterization. So, we show that we can define a function F with 4 arguments (more or less corresponding to the levels of the hierarchy) such that for all $x \in \mathbf{N}$, we have $F(\mathbf{s}^x(0), 0, 0, 0) \overset{*}{\to} F(0, 0, 0, \mathbf{s}^{\text{tower}_2(x)}(0))$ for $x > 0$.

$$F(x, y, \mathbf{s}(z), t) \to F(x, y, z, \mathbf{s}(t)) \tag{3.4}$$

$$F(x, \mathbf{s}(y), 0, t) \to F(x, y, t, t) \tag{3.5}$$

$$F(\mathbf{s}(x), 0, 0, t) \to F(x, t, 0, \mathbf{s}(0)) \tag{3.6}$$

From 3.4, we get $F(x, y, z, t) \overset{*}{\to} F(x, y, 0, z+t)$, from 3.4 and 3.5, we get $F(x, y, 0, t) \overset{*}{\to} F(x, 0, 0, 2^y * t)$ and from 3.6 and the two previous ones, we get the tower of exponentials. This program is ordered by LPO, and any interpretation of the form $(\!|F|\!)(X, Y, Z, T) = P(X)$ for some polynomial P is compatible with the rules. It is then routine to simulate a Turing machine. □

Chapitre 4

Small complexity classes

As we have already seen, the choice of the class of functions used for the interpretations plays an important role in the characterization of programs. We go a little bit further, and we focus on even smaller complexity classes of functions : LOGSPACE, NC_1 and LINSPACE. We refer the reader to [37, 40, 27] for the proofs of the theorems stated in this section.

4.1 Linear Space

The characterization of LINSPACE presented in this section has been originally published in [37]. The essential contribution is to show that we can refine the stratification made in Section 3 to cope with smaller complexity classes.

We say that a quasi-interpretation of a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ is *purely affine* whenever all constructors are additive and all function symbols have an affine interpretation.

Let F.*lpo*.*QI*.aff be the set of programs which :

- admit a proof of termination by LPO,
- a purely affine quasi-interpretation.

The following holds :

Theorem 44. *The set of functions computed by programs in F.lpo.QI.aff is exactly* LINSPACE.

With respect to the characterization of PTIME, one sees that we relax the constraint on the path ordering (using LPO instead of MPO) which allows more recurrence schema (and in particular, substitution of parameters), but we strengthened the constrainst on the quasi-intepretation (thus constraining the size of computed terms).

Theorem 45. *The set of functions computed by non-deterministic programs in F.n.lpo.QI.aff is exactly* NLINSPACE.

In the next section, we will give a precise definition of the semantics of such programs. But, briefly, following Gurevich and Grädel [67], one takes the best branch of computation, where "best" means ordered by a lexicographic ordering on values.

4.2 Cons-free programs and LOGSPACE

In this section, we reconsider the result presented in [28, 27] showing that cons-free programs (deterministic or not) characterize PTIME. This result is itself a reformulation of an older result due to Cook [49] giving a characterization of PTIME by means of pushdown automata.

Interpretations can be used to express some syntactic constrainsts. Amadio in [6] has already seen that cons-free programs have a quasi-interpretation. Simply, state that $(\!| f |\!)(x_1, \ldots, x_n) = \max(x_1, \ldots, x_n)$ for all function symbol f. However, this is not a sufficient condition. The rule $f(\mathbf{c}_1(\mathbf{c}_2(x))) \to f(\mathbf{c}_2(\mathbf{c}_1(x)))$ is compatible with any quasi-interpretation such that $(\!| \mathbf{c}_1 |\!)$ and $(\!| \mathbf{c}_2 |\!)$ are additive, but it is not cons-free. The following ordering

Let us consider a signature $\mathcal{C}$, the signature of constructors in the sequel. $S(\mathcal{C})$ denotes the set of *finite non-empty sets* of terms in $\mathcal{T}(\mathcal{C})$. On $S(\mathcal{C})$, we define $\trianglelefteq^*$ as follows : $m \trianglelefteq^* m'$ iff $\forall t \in m : \exists t' \in m' : t \trianglelefteq t'$. As an ordering on sets, for interpretations, we will use the inclusion relation. One may observe that $m \subseteq m' \Longrightarrow m \trianglelefteq^* m'$.

Definition 46. Let us consider a monotone interpretation $(\!| - |\!)$ of a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ over $(S(\mathcal{C}), \subseteq)$. We say that it preserves constructors if

1. for any constructor symbol $\mathbf{c} \in \mathcal{C}$, $(\!| \mathbf{c} |\!)(m_1, \ldots, m_k) = \{\mathbf{c}(t_1, \ldots, t_k) \mid t_i \in m_i, i = 1, \ldots, k\}$.
2. given a rule $f(p_1, \cdots, p_n) \to r$ and a ground substitution σ, for all $u \trianglelefteq r$,

$$(\!| \sigma(u) |\!) \trianglelefteq^* (\!| \sigma(f(p_1, \cdots, p_n)) |\!) \cup_{i=1}^{n} (\!| \sigma(p_i) |\!).$$

By extension, we say that a program is constructor preserving if it admits a constructor preserving monotone interpretation.

The fact that a program preserves constructors sets the definition of the interpretation over constructors. Moreover, (1) below gives a simple characterization of the interpretations of constructor terms.

Proposition 47. *Given a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ and a constructor preserving monotone interpretation $(\!| - |\!)$, the following holds*

1. *for any ground constructor term t, $(\!| t |\!) = \{t\}$,*
2. *given a ground substitution σ, for any constructor terms $u \trianglelefteq v$, $(\!| \sigma(u) |\!) \trianglelefteq^* (\!| \sigma(v) |\!)$,*
3. *if $[\![f]\!](t_1, \ldots, t_n) = t$, then $t \in (\!| f(t_1, \ldots, t_n) |\!)$.*

Definition 48. In the present context, an interpretation is said to be polynomially bounded if for any symbol f, for any sets $m_1, \ldots, m_n$, the set $(\!| f |\!)(m_1, \ldots, m_n)$ has a size polynomially bounded w.r.t. to the size of the m_i's. The size of a set m is defined to be $|m| = \sum_{t \in m} |t|$.

Example 49. Let us come back to Example 8, it has a polynomially bounded constructor preserving monotone interpretation. Apart from the generic interpretation on constructors, we define :

$$\begin{array}{rcl}
(\!| = |\!)(m, m') = (\!| \texttt{in} |\!)(m, m') & = & \{\mathbf{tt}, \mathbf{ff}\} \\
(\!| \texttt{and} |\!)(m, m') = (\!| \texttt{or} |\!)(m, m') & = & m' \cup \{\mathbf{tt}, \mathbf{ff}\} \\
(\!| \texttt{if then else} |\!)(m_b, m_y, m_z) & = & m_y \cup m_z
\end{array}$$

It is clear that this interpretation is polynomially bounded.

Actually, the notion of constructor preserving programs generalizes the notion of constructor-free programs as introduced by Jones (see for instance [89]). He has shown that tail-recursive cons-free programs characterize LOGSPACE and that cons-free programs characterize PTIME. We recall that a program is constructor-free whenever, for any rule $f(p_1, \cdots, p_n) \to r$, for any subterm $t \trianglelefteq r$,

– if t is a constructor term, then $t \trianglelefteq f(p_1, \cdots, p_n)$,
– otherwise, the root of t is not a constructor symbol.

Proposition 50. *Any* constructor-free *program* $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ *has a polynomially bounded constructor preserving monotone interpretation* $(\!|-|\!)$.

We provide an example of a program which is not constructor-free, but with a constructor preserving interpretation.

Example 51. Using the tally numbers $\mathbf{0}, \mathbf{s}$, and lists, the function $\mathtt{f}$ builds the list of the first $n-1$ integers given the argument n.

$$\begin{array}{rcl} \mathtt{f}(\mathbf{0}) & \to & \mathtt{nil} \\ \mathtt{f}(\mathbf{s}(n)) & \to & \mathbf{cons}(n, \mathtt{f}(n)) \end{array}$$

Such a program has a constructor preserving interpretation. Let

$$\begin{array}{rcl} (\!|f|\!)(m) & = & \{\mathbf{cons}(n_1, \mathbf{cons}(n_2, \cdots (\mathbf{cons}(n_k, \mathtt{nil})) \cdots)) \mid \\ & & \quad \exists n_1, \ldots, n_k \in m : \forall j \leq k-1 : n_j = \mathbf{s}(n_{j+1})\} \\ & & \cup \{\mathtt{nil}\}. \end{array}$$

It is clear that this program is not constructor-free. But, there is a stronger difference : the function computed by this program cannot be computed by *any* constructor-free program. Indeed, recall that the output of functions computed by constructor-free programs are subterms of the inputs. Since this is not the case of $\mathtt{f}$, the conclusion follows.

Let us make one last observation about the example. Actually, the interpretation is polynomially bounded. Indeed, a list of the shape

$$\mathbf{cons}(n_1, \mathbf{cons}(n_2, \cdots (\mathbf{cons}(n_k, \mathtt{nil})) \cdots))$$

is fixed by the choice of n_1 and k. Since $k \leq |n_1| \leq |m|$, there are at most $|m|^2$ such lists, each of which has a polynomial size.

Theorem 52. *Predicates computed by programs with a polynomially bounded constructor preserving interpretation are exactly* Ptime *predicates.*

4.3 Alogtime

In this section, we recall a result which appeared in [40]. It illustrates in which terms studies of recursion schema can be reused in the context of intepretation of programs.

Definition 53 (Lightweight)**.** Given a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$, a lightweight ω is a partial assignment which ranges over $\mathcal{F}$. To a given function symbol $\mathtt{f}$ of arity n it assigns a total function $\omega_{\mathtt{f}}$ from ${\mathbf{R}^+}^n$ to $\mathbf{R}^+$ which is weakly monotonic. That is

$$\forall i = 1, \ldots, n,\ x_i \geq y_i \Rightarrow \omega_{\mathtt{f}}(\ldots, x_i, \ldots) \geq \omega_{\mathtt{f}}(\ldots, y_i, \ldots)$$

A weight is a lightweight, which has the subterm property, that is

$$\forall i = 1, \ldots, n,\ \omega_{\mathtt{f}}(\ldots, x_i, \ldots) \geq x_i$$

Definition 54. Given a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$, a term $\mathsf{C}[\mathtt{g}_1(\vec{t_1}), \ldots, \mathtt{g}_r(\vec{t_r})]$ is a *fraternity* activated by $\mathtt{f}(p_1, \cdots, p_n)$ iff

1. There is a rule $\mathtt{f}(p_1, \cdots, p_n) \to \mathsf{C}[\mathtt{g}_1(\vec{t_1}), \ldots, \mathtt{g}_r(\vec{t_r})]$.
2. For each $i \leq r$, $\mathtt{g}_i \approx_{\mathcal{F}} \mathtt{f}$.
3. For every function symbol $\mathtt{h}$ that appears in the context C, we have $\mathtt{f} \succ \mathtt{h}$.

Definition 55 (Arboreal)**.** A program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ admits an *arboreal* sup-interpretation iff there is a sup-interpretation $(\!|\,|\!)$, a lightweight ω and a constant $K > 1$ such that for every fraternity $\mathsf{C}[\mathtt{g}_1(\vec{t_1}), \ldots, \mathtt{g}_r(\vec{t_r})]$ activated by $\mathtt{f}(p_1, \cdots, p_n)$, and any substitutions σ, both conditions are satisfied :

$$\omega_{\mathtt{f}}((\!|p_1\sigma|\!), \ldots, (\!|p_n\sigma|\!)) > 1 \tag{4.1}$$

$$\omega_{\mathtt{f}}((\!|p_1\sigma|\!), \ldots, (\!|p_n\sigma|\!)) \geq K \times \omega_{\mathtt{g}_i}((\!|t_{i,1}\sigma|\!), \ldots, (\!|t_{i,m}\sigma|\!)) \qquad \forall 1 \leq i \leq r \tag{4.2}$$

The constant K is called the arboreal coefficient of $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$.

Let us observe that a program admitting an arboreal sup-interpretation is terminating. Actually, the termination of an arboreal program may be established by the dependency pair method of Arts and Giesl [11], or by the size change principle for program termination of Lee, Jones and Ben-Amram [99].

Theorem 56. *Assume that the program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ admits an arboreal sup-interpretation. Then $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ is terminating.*

Definition 57. A function symbol $\mathtt{f}$ is *explicitly defined* iff for each rule $\mathtt{f}(p_1, \cdots, p_n) \to e$, the expression e is built from variables, constructors, operators and function symbols whose precedence is strictly less than $\mathtt{f}$ and which are also explicitly defined.

An expression e is explicit in $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ iff each function symbol occurring in e is explicitly defined in $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$. An explicit function is a function which is defined by a program in which any function symbols are explicitly defined.

Definition 58. A program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ is *explicitly fraternal* iff for each fraternity $\mathsf{C}[\mathtt{g}_1(\vec{t_1}), \ldots, \mathtt{g}_r(\vec{t_r})]$ of $\mathtt{f}$, the context C and each $t \in \vec{t_i}$ are explicitly defined in $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$.

A program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ admits an arboreal additive sup-interpretation iff it admits an arboreal sup-interpretation such that both the sup-interpretation and the lightweight ω are additive.

Theorem 59. *A function φ over $\{0,1\}^*$, whose growth is bounded by a polynomial in the length of the inputs, is computed by a program which*

- *admits an arboreal additive sup-interpretation and*
- *is explicitly fraternal*

iff φ is bitwise ALogTime.

The proof of this Theorem relies on arguments that were introduced by Leivant and Marion in [103]. The difficulty here comes from the fact that the syntax allowed to programs is more general.

Chapitre 5

Confluence from a complexity point of view

The issue of confluence of Term Rewriting Systems has been largely studied. Among remarkable properties, it is modular and algorithms are given to automatically compute the confluence of a program given its termination (based on Newman's Lemma [121], the algorithm is due to Knuth and Bendix [92], see also [84]).

Here, we consider an other point of view, originally presented in [28]. We would like to observe the role of non-confluence with respect to implicit complexity theory. Clearly, there is an intrinsic motivation for such a study. Since non-confluence can give us some freedom to write programs, it is of interest to observe which new functions can be computed with this new feature.

We also think that the study of non confluent programs contains informations about confluent programs. We have observed that the bigger the non-confluent class is the bigger is the corresponding confluent class with respect to intentionality. For instance, the set of F.cons-free programs contains "less" programs than F.SI.+.*mpo* which itself contains "less" programs than F.SPI.+.*mpo*.

$$\begin{array}{ccccc} \texttt{F}.\mathit{cons{-}free} & & \texttt{F}.\mathrm{SI}.+.\mathit{mpo} & & \texttt{F}.\mathrm{SPI}.+.\mathit{mpo} \\ \downarrow\simeq & & \downarrow\simeq & & \downarrow\simeq \\ \textsc{Ptime} & = & \textsc{Ptime} & = & \textsc{Ptime} \end{array}$$

For the non confluent part of the diagram, it turns out that :

$$\begin{array}{ccccc} \texttt{F}.n.\mathit{cons{-}free} & & \texttt{F}.n.\mathrm{SI}.+.\mathit{mpo} & & \texttt{F}.n.\mathrm{SPI}.+.\mathit{mpo} \\ \downarrow\simeq & & \downarrow\simeq & & \downarrow\simeq \\ \textsc{Ptime} & \subseteq & \textsc{NPtime} & \subseteq & \textsc{Pspace} \end{array}$$

It is natural to think of confluence as a form of (non)-determinism. Indeed, a program which is confluent can easily be simulated step by step by a deterministic Turing Machine and vice versa. However, for some class of programs, this observation fails. For instance, F.*n*.SPI. + .*mpo* programs compute Pspace but F.SPI. + .*mpo* (confluent) programs compute Ptime. We think once again that this mismatch reveals the intentional properties of class of programs.

5.1 Semantics

We first have to define precisely the notion of functions defined by a non-confluent program. Computations lead to several normal forms, depending on the reductions applied. At first sight, we shall consider a non-confluent rewrite system as a non-deterministic algorithm. Let us illustrate this point of view by programming satisfiability of boolean formulae.

In this section, given a program, we do not suppose its underlying rewriting system to be confluent. By extension, we say that such programs are not confluent (even if they may be so).

Example 60. A 3-SAT formula is given by a list of clauses, written $\vee(x_1, x_2, x_3)$ where the x_i have either the shape $\mathtt{n}(n_i)$ or $\mathtt{p}(n_i)$. The term $\mathtt{p}(n_i)$ corresponds to a positive occurrence of the variable x_{n_i} and $\mathtt{n}(n_i)$ to a negative one. The n_i's which are the identifiers of the variables are written in binary, with unary constructors $0, 1$ and the constant ε. For the sake of simplicity, we suppose all identifiers to have the same length. Since we restrict our attention to 3-SAT formulae, the disjunction $\vee$ is considered to be a ternary function. For instance the formula $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_1})$ is represented by :

$$\begin{aligned}&\mathbf{cons}(\vee(\mathtt{p}(0(1(\varepsilon))), \mathtt{p}(1(0(\varepsilon))), \mathtt{n}(1(1(\varepsilon)))),\\ &\quad \mathbf{cons}(\vee(\mathtt{p}(0(1(\varepsilon))), \mathtt{n}(1(0(\varepsilon))), \mathtt{n}(0(1(\varepsilon)))), \mathtt{nil})).\end{aligned}$$

Recalling rules given in Example 8, the following rules compute the satisfiability of a formula given a valuation of the variables ℓ. Let us suppose that ℓ denotes the list of variables with the valuation "true", then :

$$\begin{aligned}
\mathtt{ver}(\mathtt{nil}, \ell) &\rightarrow \mathbf{tt}\\
\mathtt{ver}(\mathbf{cons}(\vee(x_1, x_2, x_3), \psi), \ell) &\rightarrow \mathtt{and}(\mathtt{or}(\mathtt{or}(\mathtt{eval}(x_1, \ell),\\
&\qquad\qquad\qquad \mathtt{eval}(x_2, \ell)),\\
&\qquad\qquad \mathtt{eval}(x_3, \ell)),\\
&\qquad \mathtt{ver}(\psi, \ell))\\
\mathtt{eval}(\mathtt{n}(n), \ell) &\rightarrow \mathtt{not}(\mathtt{in}(n, \ell))\\
\mathtt{eval}(\mathtt{p}(n), \ell) &\rightarrow \mathtt{in}(n, \ell)
\end{aligned}$$

It is sufficient to compute the set of "true" variable. This is done by the rules :

$$\begin{aligned}
\mathtt{hyp}(\mathtt{nil}) &= \mathtt{nil}\\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1), \mathbf{b}(x_2), \mathbf{c}(x_3)), \ell)) &\rightarrow \mathtt{hyp}(\ell)\\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1), \mathbf{b}(x_2), \mathbf{c}(x_3)), \ell)) &\rightarrow \mathbf{cons}(x_i, \mathtt{hyp}(\ell))\\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1), \mathbf{b}(x_2), \mathbf{c}(x_3)), \ell)) &\rightarrow \mathbf{cons}(x_i, \mathbf{cons}(x_j, \mathtt{hyp}(\ell)))\\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1), \mathbf{b}(x_2), \mathbf{c}(x_3)), \ell)) &\rightarrow \mathbf{cons}(x_i, \mathbf{cons}(x_j, \mathbf{cons}(x_k, \mathtt{hyp}(\ell))))\\
\mathtt{f}(\psi) &\rightarrow \mathtt{ver}(\psi, \mathtt{hyp}(\psi))
\end{aligned}$$

with $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \{\neg, \mathtt{p}\}$ and $i \neq j \neq k \in \{1, 2, 3\}$. The main function is $\mathtt{f}$.

The rules involving $\mathtt{hyp}$ are not confluent, and correspond exactly to the non-deterministic choice. (By Newman's Lemma, the systems considered are not weakly confluent since they are terminating.)

Such a program has a strict interpretation, given by :

$$\begin{aligned}
(\!|\neg|\!)(x) = (\!|\mathtt{p}|\!)(x) &= x+1 \\
(\!|\vee|\!)(x_1,x_2,x_3) &= x_1+x_2+x_3+10 \\
(\!|\mathtt{eval}|\!)(x,y) &= (x+1)\times y+3 \\
(\!|\mathtt{ver}|\!)(x,y) &= (x+1)\times(y+1) \\
(\!|\mathtt{hyp}|\!)(x) &= x+1 \\
(\!|\mathtt{f}|\!)(x) &= (\!|\mathtt{ver}(x,\mathtt{hyp}(x))|\!)+1
\end{aligned}$$

together with the interpretations given in Example 23.

Our notion of computation by a non-confluent system appears in Krentel's work [93], in a different context. It seems appropriate and robust, as argued by Grädel and Gurevich [67].

Given a linear order $\prec$ on symbols, it can be extended to terms using the lexicographic ordering. We use the same notation $\prec$ for this order. Then, we say that a (partial) function $\varphi : \mathcal{T}(\mathcal{C})^m \to \mathcal{T}(\mathcal{C})$ is computed by a program $\langle \mathcal{C}, \mathcal{F}, t, \mathcal{R} \rangle$ if for all $t_1, \ldots, t_m \in \mathcal{T}(\mathcal{C})$:

$$\varphi(t_1,\ldots,t_n) \text{ is defined} \Leftrightarrow \varphi(t_1,\ldots,t_n) = \max_{\prec}\{v \mid f[t_1,\ldots,t_n] \xrightarrow{!} v\}$$

5.2 F.n-program with polynomial interpretation

The result in this section have been obtained in [30]. We show here that we get the expected result : non-confluence corresponds to non-deterministic computations. As we have seen before, confluent programs with a strict additive interpretation correspond to Ptime, the non confluent ones compute NPtime.

We recall that the interpretation of a F.n-program is said to be additive if all constructor symbols are additive. Similarly, a program `f` admits an affine (resp. a multiplicative) quasi-interpretation $(\!|_|\!)$ if constructors have affine (resp. multiplicative) interpretations.

Theorem 61.

- *Functions computed by F.n-program with an additive polynomial interpretation are exactly* NPtime *functions ;*
- *Functions computed by F.n-program with an affine polynomial interpretation are exactly* NEtime *functions, that is* Etime *functions ;*
- *Functions computed by F.n-program with an multiplicative polynomial interpretation are exactly* NE$_2$time *functions, that is* E$_2$time *functions.*

Démonstration. The proof follows from [30], by adapting the argument presented in Chapter 3. □

5.3 F.n-program with polynomial quasi-interpretation

For lpo, there is actually nothing surprising since Pspace= NPspace. The following Theorem holds.

Theorem 62. *Functions computed by non confluent programs that admit a quasi-interpretation and a* lpo *proof of termination are exactly* Pspace *functions.*

Let us recall that confluent systems with quasi-interpretations and a proof of termination by LPO compute PSPACE(cf. [38]). Thus, such systems, but without confluence, still contain PSPACE. Conversely, let us recall the main argument which shows that F.LPO.QI can be computed within PSPACE. The nesting of function calls is polynomial w.r.t. the size of arguments. To show this result, we essentially use Proposition 37 together with the following facts :

- $(\!|-|\!)$ has the sub-term property,
- and if $(\!|f|\!)(t_1,\dots,t_n)\xrightarrow{*}C[v]$ with v a constructor term, then $|v| \leq P(\max(|t_1|,\dots,|t_n|))$ for some polynomial P.

Since these two hypothesis remain correct for non-confluent programs, we can simulate the computation of such programs with an Alternating Turing Machine working in polynomial time. Thus, the computation can be done in PSPACE, according to Chandra, Kozen and Stockmeyer [43].

The following result is more surprising.

Theorem 63. *Functions computed by non confluent programs that admit a quasi-interpretation and a* MPO *proof of termination are exactly* PSPACE *functions.*

Example 64. [Quantified Boolean Formula] Let us compute the problem of the Quantified Boolean Formula. The algorithm involves two steps : a top-down one and a bottom-up one. In the first part, we span the computation to the leaves where we make an assumption on the value of (some of) the variables. In the second part, coming back to the top, we compute the truth value of the formula and we check the compatibility of the assumptions made in the different branches.

As above, we suppose that variables are represented by binary strings on constructors $0, 1, \varepsilon$. Constructors $\bullet, \color{red}{\bullet}, \circ$ are added to decorate variables. Given a variable n, the decoration gives the truth value of the variable. $\color{red}{\bullet}(n)$ corresponds to an unchosen value, that is true or false, $\circ(n)$ corresponds to false, and $\bullet(n)$ to true. The symbol $\perp$ is used for the trash. $\mathbf{T}$ and $\mathbf{F}$ are two (unary) constructors representing booleans within the computations. For the sake of clarity, we give an informal type to the key functions : Ψ corresponds to formulae, Λ to lists (of decorated variables) and B to truth values. Truth values are terms of the shape $\mathbf{T}(\Lambda)$ or $\mathbf{F}(\Lambda)$. Finally, V is the type of the variables and $\mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$. The following rules correspond to the first step of the computation. $\mathtt{f} : \Psi \to B$, $\mathtt{ver} : \Psi \times \Lambda \to B$, $\mathtt{not} : B \to B$, $\mathtt{or} : B \times B \to B$, $\mathrm{hyp} : B \times V \times \mathbb{B} \to B$, $\mathrm{put} : \Lambda \times V \times \mathbb{B} \to \Lambda$ and $\mathrm{hypList} : \Psi \to \Lambda$:

$$
\begin{array}{rcl}
\mathtt{f}(\varphi) & \to & \mathtt{ver}(\varphi, \mathrm{hypList}(\varphi)) \\
\mathtt{ver}(\mathbf{Var}(x), h) & \to & \mathbf{T}(\mathrm{put}(h, x, \mathbf{tt})) \\
\mathtt{ver}(\mathbf{Var}(x), h) & \to & \mathbf{F}(\mathrm{put}(h, x, \mathbf{ff})) \\
\mathtt{ver}(\mathbf{Or}(\varphi_1, \varphi_2), h) & \to & \mathtt{or}(\mathtt{ver}(\varphi_1, h), \mathtt{ver}(\varphi_2, h)) \\
\mathtt{ver}(\mathbf{Not}(\varphi), h) & \to & \mathtt{not}(\mathtt{ver}(\varphi, h)) \\
\mathtt{ver}(\mathbf{Exists}(x, \varphi), h) & \to & \mathtt{or}(\mathrm{hyp}(\mathtt{ver}(\varphi, h), x, \mathbf{tt}), \mathrm{hyp}(\mathtt{ver}(\varphi, h), x, \mathbf{ff}))
\end{array}
$$

The rules $\mathtt{ver}(\mathbf{Var}(x), h) \to \mathbf{T}(\mathrm{put}(h, x, \mathbf{tt}))$ and $\mathtt{ver}(\mathbf{Var}(x), h) \to \mathbf{F}(\mathrm{put}(h, x, \mathbf{ff}))$ are the unique rules responsible of the non confluence of the program. This is the step where the value of variables is actually chosen.

Suppose that x is a variable occuring in $\mathbf{Or}(\varphi_1, \varphi_2)$. One key feature is that in a computation of $\mathtt{ver}(\mathbf{Or}(\varphi_1, \varphi_2), h) \to \mathtt{or}(\mathtt{ver}(\varphi_1, h), \mathtt{ver}(\varphi_2, h))$, the choice of the truth value of the variable

x can be different in the two sub-computation $\mathtt{ver}(\varphi_1, h)$ and $\mathtt{ver}(\varphi_2, h)$. Then, the role of the bottom-up part of the computation is to verify that these choices are actually compatible.

put(h, x, b) update the value of the variable x to b in the list h. hypList(φ) returns a valuation of the variables in φ with an unchosen valuation.

$$
\begin{aligned}
\mathrm{hypList}(\mathbf{Var}(x)) &\rightarrow \varepsilon \\
\mathrm{hypList}(\mathbf{Or}(\varphi_1, \varphi_2)) &\rightarrow \mathrm{append}(\mathrm{hypList}(\varphi_1), \mathrm{hypList}(\varphi_2)) \\
\mathrm{hypList}(\mathbf{Not}(\varphi)) &\rightarrow \mathrm{hypList}(\varphi) \\
\mathrm{hypList}(\mathbf{Exists}(x, \varphi)) &\rightarrow \mathbf{cons}({\color{red}\bullet}(x), \mathrm{hypList}(\varphi)) \\
\mathrm{put}(\mathbf{cons}({\color{red}\bullet}(n), l), m, \mathbf{tt}) &\rightarrow \mathtt{if}\ n = m\ \mathtt{then}\ \mathbf{cons}(\bullet(n), l) \\
&\ \ \mathtt{else}\ \mathbf{cons}({\color{red}\bullet}(n), \mathrm{put}(l, m, \mathbf{tt})) \\
\mathrm{put}(\mathbf{cons}({\color{red}\bullet}(n), l), m, \mathbf{ff}) &\rightarrow \mathtt{if}\ n = m\ \mathtt{then}\ \mathbf{cons}(\circ(n), l) \\
&\ \ \mathtt{else}\ \mathbf{cons}({\color{red}\bullet}(n), \mathrm{put}(l, m, \mathbf{ff}))
\end{aligned}
$$

So, after the top-down part of the computations, one gets a "tree" whose interior nodes are labeled with "`or`" and "`not`". At the leaves, we have $\mathbf{T}(l)$ or $\mathbf{F}(l)$ where l stores the truth value of variables. The logical rules are :

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\mathtt{not}(\mathbf{F}(\ell)) & \rightarrow & \mathbf{T}(\ell) & \mathtt{or}(\mathbf{T}(\ell), \mathbf{T}(\ell')) & \rightarrow & \mathbf{T}(\mathrm{match}(\ell, \ell')) \\
\mathtt{not}(\mathbf{T}(\ell)) & \rightarrow & \mathbf{F}(\ell) & \mathtt{or}(\mathbf{F}(\ell), \mathbf{T}(\ell')) & \rightarrow & \mathbf{T}(\mathrm{match}(\ell, \ell')) \\
 & & & \mathtt{or}(\mathbf{T}(\ell), \mathbf{F}(\ell')) & \rightarrow & \mathbf{T}(\mathrm{match}(\ell, \ell')) \\
 & & & \mathtt{or}(\mathbf{F}(\ell), \mathbf{F}(\ell')) & \rightarrow & \mathbf{F}(\mathrm{match}(\ell, \ell'))
\end{array}
$$

where ℓ and ℓ' correspond to the list of hypothesis. The function match takes two lists and check that the truth value of variables occuring in both list are compatible. ${\color{red}\bullet}(n)$ is compatible with both $\circ(n)$ and $\bullet(n)$. But $\circ(n)$ and $\bullet(n)$ are not compatible.

$$
\begin{aligned}
\mathrm{match}(\varepsilon, \varepsilon) &\rightarrow \varepsilon \\
\mathrm{match}(\mathbf{cons}(x, l), \mathbf{cons}(x, l')) &\rightarrow \mathbf{cons}(x, \mathrm{match}(l, l')) \\
\mathrm{match}(\mathbf{cons}(\bullet(x), l), \mathbf{cons}({\color{red}\bullet}(x), l')) &\rightarrow \mathbf{cons}(\bullet(x), \mathrm{match}(l, l')) \\
\mathrm{match}(\mathbf{cons}(\circ(x), l), \mathbf{cons}({\color{red}\bullet}(x), l')) &\rightarrow \mathbf{cons}(\circ(x), \mathrm{match}(l, l')) \\
\mathrm{match}(\mathbf{cons}({\color{red}\bullet}(x), l), \mathbf{cons}(\bullet(x), l')) &\rightarrow \mathbf{cons}(\bullet(x), \mathrm{match}(l, l')) \\
\mathrm{match}(\mathbf{cons}({\color{red}\bullet}(x), l), \mathbf{cons}(\circ(x), l')) &\rightarrow \mathbf{cons}(\circ(x), \mathrm{match}(l, l')) \\
\mathrm{match}(\mathbf{cons}(\bullet(x), l), \mathbf{cons}(\circ(x), l')) &\rightarrow \perp \\
\mathrm{match}(\mathbf{cons}(\circ(x), l), \mathbf{cons}(\bullet(x), l')) &\rightarrow \perp
\end{aligned}
$$

The matching process runs only for lists of equal length and variables must be presented in the same order. This hypothesis is fulfilled for our program. The last verification corresponds to the **Exists** constructor. For the left branch of the `or`, the variable x is supposed to be true while for the second branch it is supposed to be false. This verification is performed by the hyp function.

$$\begin{array}{rcl}
\mathrm{hyp}(\mathbf{T}(h),x,y) & \to & \mathbf{T}(\mathrm{hyp}(h,x,y)) \\
\mathrm{hyp}(\mathbf{F}(h),x,y) & \to & \mathbf{F}(\mathrm{hyp}(h,x,y)) \\
\mathrm{hyp}(\mathbf{cons}(\bullet(y),l),x,\mathbf{tt}) & \to & \mathtt{if}\ x=y\ \mathtt{then}\ l \\
 & & \mathtt{else}\ \mathbf{cons}(\bullet(y),\mathrm{hyp}(l,x,\mathbf{tt})) \\
\mathrm{hyp}(\mathbf{cons}(\circ(y),l),x,\mathbf{tt}) & \to & \mathtt{if}\ x=y\ \mathtt{then}\ \bot \\
 & & \mathtt{else}\ \mathbf{cons}(\circ(y),\mathrm{hyp}(l,x,\mathbf{tt})) \\
\mathrm{hyp}(\mathbf{cons}(\circ(y),l),x,\mathbf{ff}) & \to & \mathtt{if}\ x=y\ \mathtt{then}\ l \\
 & & \mathtt{else}\ \mathbf{cons}(\circ(y),\mathrm{hyp}(l,x,\mathbf{ff})) \\
\mathrm{hyp}(\mathbf{cons}(\bullet(y),l),x,\mathbf{ff}) & \to & \mathtt{if}\ x=y\ \mathtt{then}\ \bot \\
 & & \mathtt{else}\ \mathbf{cons}(\bullet(y),\mathrm{hyp}(l,x,\mathbf{ff}))
\end{array}$$

The rules for `if then else` , for $=$ and for append are omitted.

It is then routine to verify that this program is ordered by MPO. The order $\mathtt{f} \succ \mathtt{ver} \succ \mathrm{hyp} \succ \mathrm{put} \succ \mathtt{or} \succ \mathtt{not} \succ \mathrm{hypList} \succ \mathrm{match} \succ \mathtt{if} \succ \mathrm{append} \succ =$ is compatible with the rules.

To end the Example, we provide a quasi-intepretation. $(\!| \mathtt{ver} |\!)(X,H) = X+H$, $(\!| \mathtt{f} |\!)(\Phi) = 2\Phi$, and for all other function symbols we take $(\!| f |\!)(X_1,\ldots,X_n) = \max(X_1,\ldots,X_n)$. For constructors, we set $(\!| \mathbf{c} |\!)(X_1,\ldots,X_n) = \sum_{i=1}^n X_i + 1$.

Proof of Theorem 63. Let us begin with the following proposition.

Proposition 65. *F.QI.mpo is closed by composition. In other words, if $f, g_1, \ldots g_k$ are computable with programs in F.QI.mpo, then, the function*

$$\lambda x_1,\ldots,x_n.f(g_1(x_1,\ldots,x_n),\ldots,g_k(x_1,\ldots,x_n))$$

is computable by a program in F.QI.mpo.

Démonstration. One adds a new rule $h(x_1,\ldots,x_n) \to f(g_1(x_1,\ldots,x_n),\ldots,g_k(x_1,\ldots,x_n))$ with precedence $h \succ f$ and $h \succ g_i$ for all $i \leq k$. The rule is compatible with the interpretation :

$$(\!| h |\!)(x_1,\ldots,x_n) = (\!| f(g_1(x_1,\ldots,x_n),\ldots,g_k(x_1,\ldots,x_n)) |\!).$$

□

Example 64 shows that a PSPACE-complete problem can be solved in the considered class of programs. By composition of QBF with the reduction, since polynomial time functions can be computed in F.QI.*mpo.n*, any PSPACE predicate can be computed in F.QI.*mpo.n*. Let us recall now that computing bit ith of the output of a PSPACE function is itself computable in PSPACE. Since building the list of the first integers below some polynomials can be computed in polynomial time, by composition, the conclusion follows. □

5.4 Non confluent constructor preserving programs

Theorem 66. *Predicates computed by non confluent programs with a polynomially bounded constructor preserving interpretation are exactly the* PTIME*-computable predicates.*

This result is close to the one of Cook in [49] (Theorem 2). He gives a characterization of PTIME by means of auxiliary pushdown automata working in logspace, that is a Turing Machine working in logarithmic space plus an extra (unbounded) stack. Moreover, the result holds whether or not the auxiliary pushdown automata is deterministic.

As the proof follows the lines of [27], we simply give a sketch of the proof. The key observation is that arguments of recursive calls are sub-terms of the initial interpretation, a property that holds for non confluent programs. As a consequence, following a call-by-value semantics, any arguments in sub-computations are some sub-terms of the initial interpretations. From that, it is possible to use memoization, see [88]. The original point is that we have to manage non-determinism.

The proof of Proposition 47 holds for non confluent programs. So, normal forms of a term t are in $(\!| t |\!)$. As we have seen in the proof of Theorem 52, this set has a polynomial size w.r.t. the size of the inputs. In the non confluent case, the cache is still a map, with the same keys, but the values of the map enumerate the list of normal forms. With the previous observation, the list of these normal forms remain polynomial, and thus, we can state that the map still has a polynomial size.

Chapitre 6

Interpretations over the reals

This chapter is devoted to interpretations over the reals, it is based on a paper [31] presented in 2010 at the TAMC conference. In particular, one will find the proofs of the main statements of this section.

One of the interesting points to choose real numbers rather than natural numbers as a universe of interpretations is that we get (at least from a theoretical point of view) a procedure to verify the validity of an interpretation of a TRS by Tarski's decomposition procedure [135]. Furthermore, we can even give a semi-algorithm to compute interpretations : indeed, for a given choice of the degree of the interpretations, finding the coefficients of these polynomial becomes decidable. Actually, since the problem of the verification or the synthesis (up to some degree) can be stated by a first order formula with 2 alternations of quantifiers, following Roy et al. [19], the complexity of these algorithms is exponential with respect to the size of the TRS.

A second advantage of the use of reals versus integers lies in the fact that it enlarges the set of rewrite systems that have an interpretation as recently shown by Lucas [105]. The point is a little bit controversial (see [120]) : it depends on the exact definition of which function can be used for interpretations. Anyway, such a discussion goes far beyond the present issue.

However, the order over reals is not well-founded. Consequently, all the complexity results obtained so far must then be reconsidered.

6.1 Synthesis and Verification of interpretations over the reals

Results shown in this Section have been published in [39]. Here, we suppose that interpretations are done over the set of positive reals $\mathbf{R}^+$.

Definition 67 (Synthesis)**.** Given a class of functions $\mathcal{F}$, the *synthesis problem* is as follows.

inputs : A program `f`.

problem : Does there exist an $\mathcal{F}$-assignment $(\!|-|\!)$ which is an interpretation for `f` ?

The verification problem consists in :

Definition 68 (Verification)**.**

inputs : A program `f` and an assignment $(\!|-|\!)$.

problem : Is the assignment $(\!|-|\!)$ an interpretation for `f` ?

Theorem 69 (Bonfante, Marion, Moyen, Péchoux [39])**.** *For strict interpretations, quasi-interpretations and monotone interpretations, the synthesis problem for* ***Max-Poly*** *assignments of bounded* $\times$*-degree and bounded* max*-degree is decidable in double exponential time in the size of the program.*

Démonstration. A complete proof is given in [39]. It is based on the fact that the properties (SMon), (WMon), (WSub), (SCmp), (WCmp), Cons can be expressed as first order formulae with an alternation of 2 quantifiers. However, (NFA) involves the rewriting relation. Therefore, there is no procedure to compute sup-interpretations in general. □

A possible approach to find interpretations at lower cost is to restrict the set of interpretations. For instance, one can restrict interpretations to the set of **Max-Plus** function. In other words, interpretations are of the form $\max_{i \in I}(\sum_{j=1..n} a_{i,j}x_j + b_i)$. We say that a polynomial is of degree (k,d) if I has less than k elements and $a_{i,j} \leq d$ for all $i \in I$ and j.

Then, given k and d, Amadio showed in [5] that the problem of the synthesis of (k,d)-interpretation, restricted to natural numbers, can be formulated as a $\exists\forall$ formula of Presburger Arithmetic. Consequently, it is decidable. Actually, it is NP-hard [5].

In [39], it is shown that the problem remains NP-hard when considered on real numbers. Actually, it is in NP and consequently NP-complete.

Theorem 70 ([39]). *The synthesis problem for quasi-interpretation with* ***Max-Plus*** *over the reals is NP-complete.*

Several methods for the verification of interpretations have been proposed so far, among which we note the seminal work of Lescanne [22] followed by Steinbach [133] and then Giesl [62]. They introduce some partial orders on polynomials which can be decided at low computational cost. Hong and Jakuš have unified these approaches.

Definition 71. Given a polynomial P, we say that it is positive from μ iff $\forall X_1 \geq \mu, \ldots, X_n \geq \mu : P(X_1, \ldots X_n) > 0$. It is absolutely positive from μ iff P is positive from μ and every partial deivative P^* is non negative from μ, that is $\forall X_1 \geq \mu, \ldots, X_n \geq \mu : P^*(X_1, \ldots X_n) \geq 0$.

Given a polynomial P and $\mu \in R$, the polynomial $\sigma_\mu(P)(X_1, \ldots, X_n) = P(X_1 + \mu, \ldots, X_n + \mu)$.

Theorem 72 (Hong-Jakuš [83]). *Let P be a polynomial. Then P is absolutely positive from μ if all coefficients of $\sigma_\mu(P)$ are positive and the constant term is non zero.*

The verification is then very fast, in linear time w.r.t. the size of the formula. This is one of the core procedure of the software CROCUS.

But the main concern of this Section is to show that the structure of polynomials over the reals has an important role from the point of view of complexity. Our thesis is that, in the field of complexity, due to Positivstellensatz, polynomials over the reals can safely replace polynomials over the integers. In some way, we follow the claim of Lucas in [106] who states that the field of algebraic geometry may give some new insight on issues about polynomial interpretations over the reals. The mathematical structure of polynomials over the reals has also been considered by Shkaravska et al [131] to bound the size of computed terms.

Here, we propose the application of Stengle's Positivstellensatz [134]. This deep result deals with the algebraic structure of polynomials over the reals. The key point of this section is to show that this structure has some fundamental consequences from the point of view of the complexity of programs.

6.2 Positivstellensatz and applications

In this section, we introduce a deep mathematical result, the Positivstellensatz. Then we give some direct applications to polynomial interpretations. These ingredients will play a key role in our analysis of the role of reals in complexity (see Theorems 79 and 90).

Let $n > 0$. Denote by $\mathbf{R}[x_1, \dots, x_n]$ the $\mathbf{R}$-algebra of polynomials with real coefficients. Denote by $(\mathbf{R}^+)^n = \{\vec{x} = (x_1, \dots, x_n) \in \mathbf{R}^n \mid x_1, \dots, x_n > 0\}$ the first quadrant. Since we only need to consider the $\mathbf{R}$-algebra of polynomial functions $(\mathbf{R}^+)^n \to \mathbf{R}$, it will be convenient to identify the two spaces. In particular throughout this section, all polynomial functions are defined on $(\mathbf{R}^+)^n$.

Theorem 73 (Positivstellensatz, Stengle [134])**.** *Suppose that we are given polynomials $P_1, \dots, P_m \in \mathbf{R}[x_1, \dots, x_k]$. Then, the following two assertions are equivalent :*

1. $\{x_1, \dots, x_k : P_1(x_1, \dots, x_k) \geq 0 \wedge \cdots \wedge P_m(x_1, \dots, x_k) \geq 0\} = \emptyset$
2. $\exists Q_1, \dots, Q_m : -1 = \sum_{i \leq m} Q_i P_i$ *where each Q_i is a sum of squares of polynomials (and so is positive and weakly monotonic).*

Moreover, these polynomials $Q_1, \dots, Q_m$ can be effectively computed. We refer the reader to the work of Lombardi, Coste and Roy [50]. As a consequence, all the constructions given below can be actually (at least theoretically) computed.

It will be convenient to derive from the Positivstellensatz some useful propositions for our applications.

Proposition 74. *Suppose that a TRS (Σ, R) admits an interpretation $(\!|-|\!)$ over* Max-Poly *such that for all rules $\ell \to r$, we have $(\!|\ell|\!) > (\!|r|\!)$. There exists a positive, monotonic polynomial P such that for any rule $\ell \to r$, we have $(\!|\ell|\!)(x_1, \dots, x_k) - (\!|r|\!)(x_1, \dots, x_k) \geq \dfrac{1}{P(x_1, \dots, x_k)}$.*

The proof is a direct consequence of Theorem 73 when ℓ and r are polynomials. By a finite case analysis, one may cope with the max function. One may notice that there does not exist, a priori, a constant $\delta > 0$ such that $(\!|\ell|\!)(x_1, \dots, x_k) \geq (\!|r|\!)(x_1, \dots, x_k) + \delta$. Indeed, suppose that $(\!|\ell|\!)(x, y) - (\!|r|\!)(x, y) = P(x, y) = (xy - 1)^2 + x^2$. On $\mathbf{R}^+$, $P(x, y) > 0$. On the other hand, one may observe that $\lim_{x \to 0} P(x, 1/x) = \lim_{x \to 0} x^2 = 0$. To deal with termination or complexity, in their work, Lucas (see for instance [106]) or Marion and Péchoux [109] suppose the existence of such a constant.

However, set $Q(x, y) = (1+x+y)^2$. As stated by the theorem, the inequality $P(x, y)Q(x, y) \geq 1$ holds for all $x, y \geq 0$.

Proposition 74 has an important consequence for interpretations verifying the sub-term property (that is strict and quasi-interpretations). Since, in a derivation all terms have an interpretation bounded by the interpretation of the first term, there is a minimal decay for each application of a rule of the derivation. Then, due to the next Theorem, the result can be extended to context of strict interpretations.

Theorem 75. *Given a polynomial $P \in \mathbf{R}[x_1, \dots, x_n]$ such that*

(i) $\forall x_1 \geq 0, \dots, x_n \geq 0 : P(x_1, \dots, x_n) > \max(x_1, \dots, x_n)$,

(ii) $\forall x_1 \geq 0, \dots, x_n \geq 0 : \dfrac{\partial P}{\partial x_i}(x_1, \dots, x_n) > 0$ *for all* $i \leq n$,

then, there exist $A > 0$ such that for any $\Delta > 0$, we have $P(x_1, \dots, x_i{+}\Delta, \dots, x_n) > P(x_1, \dots, x_n) + \Delta$ whenever $||\vec{x}|| > A$.

In other words, we recovered equations analogous to Equations (6.2), (6.3) for sufficiently large terms. Another application of Positivstellensatz is the following :

Proposition 76. *Suppose that a TRS (Σ, R) admits a strict interpretation $(\!|-|\!)$ over* Max-Poly*. Then, for all $A > 0$, the set of terms $\{t \in \mathcal{T}(\Sigma) \mid (\!|t|\!) < A\}$ is finite.*

In other words, there are only finitely many terms with an interpretation bounded by some arbitrary constant. Then, Theorem 75 may be applied for all but finitely many terms.

Proposition 77. *Suppose that a TRS (Σ, R) admits a strict interpretation $(\!|-|\!)$ over Poly. Then, there exists a real $A > 0$ and a positive, monotonic polynomial P such that for all $x_1, \dots, x_n \geq 0$, if $x_{i_1}, \dots, x_{i_k} > A$, then for all symbols f, we have $(\!|f|\!)(x_1, \dots, x_n) \geq x_{i_1} + \cdots + x_{i_k} + \dfrac{1}{P((\!|f|\!)(x_1, \dots, x_n))}$.*

This latter result gives (more or less) directly a bound on the size of the terms. It is a consequence of Theorem 73 and the following Theorem.

Theorem 78. *Given a polynomial $P \in \mathbf{R}[x_1, \dots, x_n]$ such that*

(i) $\forall x_1, \dots, x_n \geq 0 : P(x_1, \dots, x_n) > \max(x_1, \dots, x_n)$,

(ii) $\forall x'_i > x_i, x_1, \dots, x_n \geq 0 : P(x_1, \dots, x'_i, x_{i+1}, \dots, x_n) > P(x_1, \dots, x_n)$,

then, there exist $A \geq 0$ such that $P(x_1, \dots, x_n) > x_1 + \cdots + x_n$ whenever $||\vec{x}|| > A$.

All the hypotheses are necessary. If $P(x, y)$ is not supposed to be greater than $\max(x, y)$, you can simply take $P(x, y) = (x + y)/2$. It is strictly monotonic, but clearly, $P(x, y) < x + y$ for all $x, y > 0$.

The second hypothesis is also necessary. A counter example is given by $P(x, y) = 16(x - y)^2 + (3/2)x + 1$.

6.3 The role of reals in complexity

As already mentioned in Section 2, given a strict interpretation for a TRS, it follows immediately that for all $s \to t$, $(\!|s|\!) > (\!|t|\!)$. If one takes the interpretation on natural numbers, this leads to a bound on the derivation height.

$$\mathtt{dh}(t) \leq (\!|t|\!). \tag{6.1}$$

Indeed, suppose $t_1 \to t_2 \to \cdots \to t_n$, then $(\!|t_1|\!) > (\!|t_2|\!) > \cdots > (\!|t_n|\!)$. On natural numbers, this means that $n \leq (\!|t_1|\!)$, a conclusion that does not hold for interpretation over the reals.

Equation (6.1) is a consequence of the fact that $(\!|t|\!) \geq (\!|u|\!) + 1$ for terms $t \to u$. This fact itself is due to a) $(\!|\ell|\!) > (\!|r|\!)$ implies that

$$(\!|\ell|\!) \quad \geq \quad (\!|r|\!) + 1 \tag{6.2}$$

and b) that for all $x_i > y_i$:

$$(\!|f|\!)(x_1, \dots, x_i, \dots, x_n) - (\!|f|\!)(x_1, \dots, y_i, \dots, x_n) \geq x_i - y_i. \tag{6.3}$$

The two inequalities (6.2), (6.3) do not hold in general for real interpretations. However, we have all the tools to prove that reals can safely replace integers from the point of view of complexity theory. This is illustrated by Theorem 79 and Theorem 90.

Theorem 79. *Functions computed by programs with an additive strict interpretation (over the reals) are exactly* Ptime *functions.*

The rest of the section is devoted to (a sketch of) the proof of the Theorem 79. The main difficulty of the proof is that inequalities as given by the preceding section hold only for sufficiently large values. So, the main issue is to separate "small" terms (and "small rewriting steps") from "large" ones. Positivstellensatz gives us the arguments for large terms (Lemma 80), Lemmas 82 and 83 show that there are not too many small steps between two large steps. Lemma 85 describe how small steps and big steps alternate.

From now on, we suppose we are given a program with an additive strict interpretation over polynomials. The following Lemmas, which are direct applications of Proposition 74, 77, are the main steps to prove both Theorem 79 and Theorem 90. A full proof of these lemmas can be found in [31].

Lemma 80. *There exists a polynomial P and a real number $A > 0$ such that for all steps $\ell\sigma \to r\sigma$ with $(\!| r\sigma |\!) > A$, then, for all contexts C, we have $(\!| C[\ell\sigma] |\!) \geq (\!| C[r\sigma] |\!) + \dfrac{1}{P((\!| \ell\sigma |\!))}$.*

Démonstration. This is a consequence of Proposition 74. □

Definition 81. Given a real $A > 0$, we say that the A-size of a closed term t is the number of subterms u of t (including itself) such that $(\!| u |\!) > A$. We note $|t|_A$ the A-size of t.

Lemma 82. *There exists a real number $A > 0$ such that for all $C > A$, there is a polynomial Q for which $|t|_C \leq Q((\!| t |\!))$ for all closed terms t.*

Démonstration. This is consequence of Proposition 77.

□

For $A > 0$, we say that $t = C[\ell\sigma] \to C[r\sigma] = u$ is an A-step whenever $(\!| r\sigma |\!) > A$. We note such a rewriting step $t \to_{>A} u$. Otherwise, it is an $\leq A$-step, and we note it $t \to_{\leq A} u$. We use the usual $*$ notation for transitive closure. When we restrict the relation to the call by value strategy [9], we add "cbv" as a subscript. Let us emphasize that an $\to_{\leq A}$-normal form is not necessarily a normal form for $\to$.

Lemma 83. *There exists a constant A such that for all $C > A$ there exists a (monotonic) polynomial P such that for all terms t, any call by value derivation $t \to^*_{\leq C,cbv} u$ has length less than $P((\!| t |\!))$.*

Démonstration. This is a consequence of Proposition 74. □

Lemma 84. *For constructor terms, we have $(\!| t |\!) \leq \Gamma \times |t|$ for some constant Γ.*

Démonstration. Take $\Gamma = \max\{\dfrac{1}{\gamma_c} \mid (\!| c |\!)(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i + \gamma_c\}$. By induction on terms. □

Lemma 85. *Let us suppose we are given a program with an additive strict interpretation. There is a strategy such that for all function symbol f, for all constructor terms $t_1, \ldots, t_n$, any derivation following the strategy starting from $f(t_1, \ldots, t_n)$ has length bounded by $Q(\max(|t_1|, \ldots, |t_n|))$ where Q is a polynomial.*

9. Innermost in the present context.

Démonstration. Let us consider A as defined in Lemma 83, B and P_1 as defined in Lemma 80. We define $C = \max(A, B)$. Let P_0 be the polynomial induced from Lemma 83. Finally, let us consider the strategy as introduced above : rewrite as long as possible the according to $\rightarrow_{\leq C,\mathrm{cbv}}$, and then, apply a C-step. That is, we have $t_1 \rightarrow^*_{\leq C,\mathrm{cbv}} t'_1 \rightarrow_{>C,\mathrm{cbv}} t_2 \rightarrow^*_{\leq C,\mathrm{cbv}} t'_2 \rightarrow^* \cdots$. According to Lemma 83, there are at most $P_0((\!|t_i|\!))$ steps in the derivation $t_i \rightarrow^*_{\leq C,\mathrm{cbv}} t'_i$. From Lemma 80, we can state that there are at most $(\!|t_1|\!) \times P_1((\!|t_1|\!))$ such C-steps. Consequently, the derivation length is bounded by $(\!|t_1|\!) \times P_1((\!|t_1|\!)) \times P_0((\!|t_1|\!))$ since for all $i \geq 1$, $(\!|t_i|\!) \leq (\!|t_1|\!)$.

Consider now a function symbol $f \in \mathcal{F}$, from Lemma 84, $(\!|f(t_1,\ldots,t_n)|\!) = (\!|f|\!)((\!|t_1|\!),\ldots,(\!|t_n|\!)) \leq (\!|f|\!)(\Gamma \max(|t_1|,\ldots,|t_n|),\ldots,\Gamma \max(|t_1|,\ldots,|t_n|))$. The conclusion is immediate. □

Proof of Theorem 79. With the strategy defined above, we have seen that the derivation length of a term $f(t_1,\ldots,t_n)$ is polynomial w.r.t. $\max(|t_1|,\ldots,|t_n|)$. The computation can be done in polynomial time due to dal Lago and Martini, see [52], together with the fact that the normal form has polynomial size (Lemma 84). For the converse part, we refer the reader to [30] where a proof that PTIME programs can be computed by functional programs with strict interpretations over the integers. This proof can be safely used in the present context. □

6.4 Dependency Pairs with polynomial interpretation over the reals

Termination by Dependency Pairs is a general method introduced by Arts and Giesl [11] enlighting recursive calls. Suppose $f(t_1,\ldots,t_n) \rightarrow C[g(u_1,\ldots,u_m)]$ is a rule of the program. Then, $(F(t_1,\ldots,t_n), G(u_1,\ldots,u_m))$ is a dependency pair where F and G are new symbols associated to f and g respectively. $S(\mathcal{C},\mathcal{F},R)$ denotes the program obtained by adding these rules. The dependency graph connects dependency pairs $(u,v) \rightarrow (u',v')$ if there is a substitution σ such that $\sigma(v) \overset{*}{\rightarrow} \sigma(u)$ and termination is obtained when there is no cycles in the graph. Since the definition of the graph involves the rewriting relation, its computation is undecidable. In practice, one gives an approximation of the graph which is bigger. Since this is not the issue here, we suppose that we have a procedure to compute this supergraph which we call the dependency graph.

We mention here the work of Hirokawa and Moser [77] who gives an other characterization of PTIME by means of dependency pairs over polynomial interpretations.

Theorem 86. *[Arts,Giesl [11]] A TRS $(\mathcal{C},\mathcal{F},t,\mathcal{R})$ is terminating iff there exists a well-founded weakly monotonic quasi-ordering $\geq$, where both $\geq$ and $>$ are closed under substitution, such that*

- *$\ell \geq r$ for all rules $\ell \rightarrow r$,*
- *$s \geq t$ for all dependency pairs (s,t) on a cycle of the dependency graph and*
- *$s > t$ for at least one dependency pair on each cycle of the graph.*

It is natural to use sup interpretations for the quasi-ordering and the ordering of terms. However, the ordering $>$ is not well-founded on $\mathbf{R}$, so that system may not terminate. Here is such an example.

Example 87. Consider the non terminating system :

$$\begin{pmatrix} \mathtt{f}(\mathbf{0}) & \rightarrow & \mathbf{0} \\ \mathtt{f}(x) & \rightarrow & \mathtt{f}(\mathbf{s}(x)) \end{pmatrix}$$

Take $(\!|0|\!) = 1$, $(\!|\mathbf{s}|\!)(x) = x/2$. There is a unique dependency pair $F(x) \rightarrow F(\mathbf{s}(x))$. We define $(\!|F|\!)(x) = (\!|f|\!)(x) = x + 1$.

One way to avoid these infinite descent is to force the inequalities over reals to be of the form $P(x_1, \ldots, x_n) \geq Q(x_1, \ldots, x_n) + \delta$ for some $\delta > 0$ (see for instance Lucas's work [106] or Marion and Péchoux [109]). Doing so, one gets a well-founded partial ordering on reals. We propose an alternative approach to that problem, keeping the original ordering of $\mathbf{R}$.

Definition 88. A $\mathbf{R}$-DP-interpretation for a program associates to each symbol f a monotonic function $(\!| f |\!)$such that

1. constructors have additive interpretations,
2. $(\!| \ell |\!) \geq (\!| r |\!)$ for $\ell \to r \in R$,
3. $(\!| s |\!) \geq (\!| r |\!)$ for $(s, r) \in DP(R)$,
4. for each dependency pair (s, t) in a cycle, $(\!| s |\!) > (\!| r |\!)$ holds.

Example 89. The Quantified Boolean Formula (QBF) problem is well known to be PSPACE complete. It consists in determining the validity of a boolean formula with quantifiers over propositional variables. Without loss of generality, we restrict formulae to the constructors $\neg, \vee, \exists$. Variables are represented by strings supposed to be of equal length. The QBF problem is solved extending the program of Example 8 with :

$$\begin{aligned}
\mathtt{ver}_{\mathtt{qbf}}(\mathbf{Var}(x), t) &\to \mathtt{in}(x, t) \\
\mathtt{ver}_{\mathtt{qbf}}(\mathbf{Not}(\varphi), t) &\to \mathtt{not}(\mathtt{ver}_{\mathtt{qbf}}(\varphi, t)) \\
\mathtt{ver}_{\mathtt{qbf}}(\mathbf{Or}(\varphi_1, \varphi_2), t) &\to \mathtt{or}(\mathtt{ver}_{\mathtt{qbf}}(\varphi_1, t), \mathtt{ver}_{\mathtt{qbf}}(\varphi_2, t)) \\
\mathtt{ver}_{\mathtt{qbf}}(\mathbf{Exists}(n, \varphi), t) &\to \mathtt{or}(\mathtt{ver}_{\mathtt{qbf}}(\varphi, \mathbf{cons}(n, t)), \mathtt{ver}_{\mathtt{qbf}}(\varphi, t)) \\
\mathtt{qbf}(\varphi) &\to \mathtt{ver}_{\mathtt{qbf}}(\varphi, \varepsilon)
\end{aligned}$$

The QBF problem can be given a $\mathbf{R}$-DP interpretation :

$$\begin{aligned}
(\!| \mathbf{Not} |\!)(x) = (\!| \mathbf{Var} |\!)(x) &= x + 1 \\
(\!| \mathbf{Or} |\!)(x, y) = (\!| \mathbf{Exists} |\!)(x, y) &= x + y + 1 \\
(\!| \mathtt{ver}_{\mathtt{qbf}} |\!)(x, y) = (\!| \mathtt{qbf} |\!)(x) &= 1 \\
(\!| \mathrm{NOT} |\!)(x) &= x \\
(\!| \mathrm{OR} |\!)(x, y) = (\!| \mathrm{EQ} |\!)(x, y) &= \max(x, y) \\
(\!| \mathrm{IN} |\!)(x, y) &= x + y \\
(\!| \mathrm{VERIFY} |\!)(x, y) &= 2x + y + 1 \\
(\!| \mathrm{QBF} |\!)(x) &= 2x + 1
\end{aligned}$$

Theorem 90. *Functions computed by programs*
– *with additive* $\mathbf{R}$*-DP-interpretations*
– *the interpretation of any capital symbol* F *has the sub-term property*
are exactly PSPACE *computable functions.*

Démonstration. The completeness comes from the example of the QBF, together with the compositionality of such interpretation.

Conversely, the key argument is to prove that the call tree has a polynomial depth w.r.t. the size of arguments. The proof relies again on Lemmas 80, 82, 83, 85 adapted to dependency pairs (in cycles). Indeed, since capital symbol have the sub-term property, the lemmas are actually valid in the present context.[10] The rewriting steps of dependency pairs can be reinterpreted

10. Lemma 82 is immediate here since we focus on terms of the shape $F(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are constructor terms.

as depth-first traversal in the call-tree. Thus, we can state that the depth of the call tree is polynomial, as we stated in an analogous way that the derivation length was polynomial. The function can be computed by an Alternating Turing Machine working in polynomial time, thus by a Turing Machine working in polynomial space. □

Chapitre 7

From Term Rewriting to polygraphs

In this chapter, we present polygraphs, a model of computation whose objects are graphs as opposed to terms. Our interest concerns the consequences of the choice of using graphs with respect to complexity. Many formalisms could have been considered. Indeed, there are many different computational models for which graphs are first-class citizens. Among them, we note the bigraph of Milner [112], but also some models issued from the bio/chemical paradigms such as [18, 9, 124, 23]. All the result presented in this section already appeared in [33].

7.1 A first glance at polygraphs

As a first approach, one can consider polygraphs as rewriting systems on algebraic circuits, made of :

Types − They are the wires, called 1-cells. Each one conveys information of some elementary type. To represent product types, one uses several wires, in parallel, calling such a construction a 1-path. For example, the following 1-path represents the type of quadruples made of an integer, a boolean, a real number and a boolean :

int bool real bool

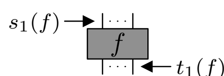
The 1-paths can be composed in one way, by putting them in parallel :

$$u \star_0 v = u \; v$$

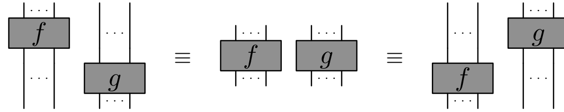
Operations − They are represented by circuits, called 2-paths. The gates used to build them are called 2-cells. The 2-paths can be composed in two ways, either by juxtaposition (parallel composition) or by connection (sequential composition) :

$$f \star_0 g = f \; g \qquad\qquad f \star_1 g = \begin{matrix} f \\ g \end{matrix}$$

Each 2-path (or 2-cell) has a finite number of typed inputs, a 1-path called its 1-source, and a finite number of typed outputs, a 1-path called its 1-target :



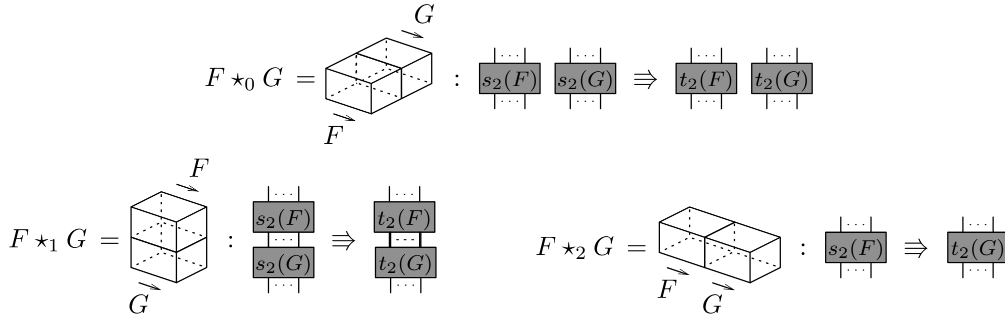
Several constructions represent the same operation. In particular, wires can be stretched or contracted, provided one does not cross them or break them. This can be written either graphically or algebraically :



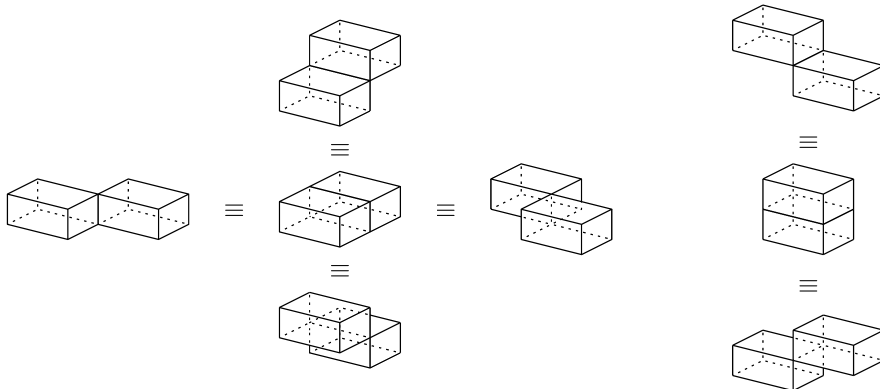
$$\big(f \star_0 s_1(g)\big) \star_1 \big(t_1(f) \star_0 g\big) \quad \equiv \quad f \star_0 g \quad \equiv \quad \big(s_1(f) \star_0 g\big) \star_1 \big(f \star_0 t_1(g)\big).$$

Computations – They are rewriting paths, called 3-paths, transforming a given 2-path, called its 2-source, into another one, called its 2-target. The 3-paths are generated by local rewriting rules, called 3-cells. The 2-source and the 2-target of a 3-cell or 3-path are required to have the same input and output, *i.e.* the same 1-source and the same 1-target. A 3-path is represented either as a reduction on 2-paths or as a genuine 3-dimensional object :

The 3-paths can be composed in three ways, two parallel ones coming from the structure of the 2-paths, plus one new, sequential one :



The 3-paths are identified modulo relations that include topological moves such as :



These graphical relations have an algebraic version given, for $0 \leq i < j \leq 2$, by :

$$\big(F \star_i s_j(G)\big) \star_j \big(t_j(F) \star_i G\big) \quad \equiv \quad F \star_i G \quad \equiv \quad \big(s_j(F) \star_i G\big) \star_j \big(F \star_i t_j(G)\big).$$

So far, we have described a special case of 3-polygraphs. A n-polygraph is a similar object, made of cells, paths, sources, targets and compositions in all dimensions up to n.

Given some cells $\mathcal{P}$, $\langle\mathcal{P}\rangle$ denotes the set of path built from $\mathcal{P}$. The empty context (it is actually the 0-cell) is denoted by $*$. The size of a 1-path (resp. 2-path, 3-path) t, denoted by $|t|$ (resp. $||t||$, $|||t|||$) is the number of 1-cells (resp. 2-cells, 3-cells) occurring in t. Given some set of 1-cells X, $|t|$ denotes the number of occurrences of 1-cells in t restricted to X. The notations $||t||_X$ and $|||t|||_X$ are used respectively for 2-path and 3-path.

Remark 91. Polygraphs provide a uniform, algebraic and graphical description of objects coming from different domains : abstract, string and term rewriting systems [96, 71], abstract algebraic structures [42, 70, 104], Feynman and Penrose diagrams [14], braids, knots and tangles diagrams equipped with the Reidemeister moves [1, 70], Petri nets [73] and propositional proofs of classical and linear logics [72].

7.2 Polygraphic programs

A program is roughly speaking a 3-polygraph. As it has been done for terms, one can distinguish constructor cells from "function" cells. We supply polygraphic programs with structure cells, some of them to erase some values, some to duplicate values, and the last ones for the permutations. From a traditional programming perspective, erasing, duplication and permutations are given for free in the syntax. This is not the case with polygraph. So, to show that the programmer does not have to take care of such manipulations, we provide an explicit management of structure cells.

The letter "C" serves for "constructors", the letter "F" for function and "S" is for structure,

Definition 92. A *polygraphic program* is a finite 3-polygraph $\mathcal{P}$ with one 0-cell, thereafter denoted by $*$, and such that its sets of 2-cells and of 3-cells respectively decompose into $\mathcal{P}_2 = \mathcal{P}_2^S \cup \mathcal{P}_2^C \cup \mathcal{P}_2^F$ and $\mathcal{P}_3 = \mathcal{P}_3^S \cup \mathcal{P}_3^R$, with the following conditions :

- The set $\mathcal{P}_2^S$ is made of the following elements, called *structure 2-cells*, where ξ and ζ range over the set of 1-cells of $\mathcal{P}$:

$$\triangle_\xi : \xi \Rightarrow \xi \star_0 \xi, \qquad \bullet_\xi : \xi \Rightarrow *.$$

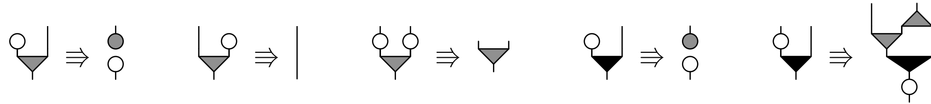
When the context is clear, one simply writes $\times$, $\triangle$ and $\bullet$. The following elements of $\langle\mathcal{P}_2^S\rangle$ are called *structure* 2-*paths* and they are defined by structural induction on their 1-source :

- The set $\mathcal{P}_2^C$ is made of 2-cells with coarity 1, *i.e.* of the shape ∇, called *constructor 2-cells.*
- The elements of $\mathcal{P}_2^F$ are called *function 2-cells.*
- The elements of $\mathcal{P}_3^S$, called *structure 3-cells*, are defined, for every constructor 2-cell $\nabla : x \Rightarrow \xi$ and every 1-cell ζ, by :

- The elements of $\mathcal{P}_3^R$ are called *computation 3-cells* and each one has a 2-source of the shape $t \star_1 \square$, with $t \in \langle\mathcal{P}_2^C\rangle$ and $\square \in \mathcal{P}_2^F$.

Example 93. The following polygraphic program $\mathcal{D}$ computes the euclidean division on natural numbers (a precise definition will be given later) :

1. It has one 1-cell `n`, standing for the type of natural numbers.
2. Apart from the fixed three structure 2-cells, it has two constructor 2-cells, $: * \Rightarrow$ `n` for zero and $:$ `n` $\Rightarrow$ `n` for the successor operation, and two function 2-cells, $:$ `n` $\star_0$ `n` $\Rightarrow$ `n` for the minus function and $:$ `n` $\star_0$ `n` $\Rightarrow$ `n` for the division function.
3. Its 3-cells are made of eight structure 3-cells, plus the following five computation 3-cells :



Example 94. The following program $\mathcal{F}$ computes the *fusion sort* function on lists of natural numbers lower or equal than some constant $N \in \mathbb{N}$:

1. Its 1-cells are `n`, for natural numbers, and `l`, for lists of natural numbers.
2. Its 2-cells are made of eight structure 2-cells, plus :

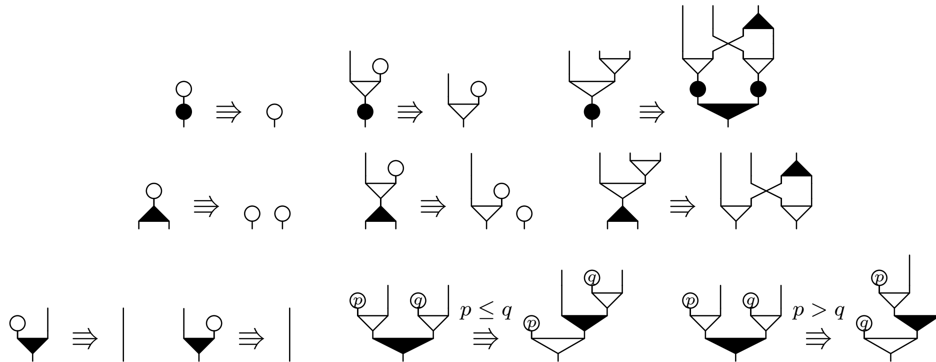
   (a) Constructor 2-cells, for the natural numbers 0, ..., N, the empty list and the list constructor :

   $$\left(: * \Rightarrow \mathtt{n} \right)_{0 \leq n \leq N}, \qquad : * \Rightarrow \mathtt{l}, \qquad : \mathtt{n} \star_0 \mathtt{l} \Rightarrow \mathtt{l}.$$

   (b) Function 2-cells, respectively for the main sort and the two auxiliary split and merge :

   $$: \mathtt{l} \Rightarrow \mathtt{l}, \qquad : \mathtt{l} \Rightarrow \mathtt{l} *_0 \mathtt{l}, \qquad : \mathtt{l} *_0 \mathtt{l} \Rightarrow \mathtt{l}.$$

3. Its 3-cells are made of $6N + 18$ structure 3-cells, plus $N^2 + 2N + 8$ computation 3-cells :



Remark 95. One may object that sorting lists when the a priori bound N is known can be performed in a linear number of steps : one reads the list and counts the number of occurrences of each element, then produces the sorted list from this information. Nevertheless, the presented algorithm (up to the test $\leq$ on the natural numbers p and q) really mimics the "mechanics" of the fusion sort algorithm.

Why don't we internalize the comparison of numbers within the polygraphic program ? This comes from the fact that the *if-then-else* construction implicitly involves an evaluation strategy : one first computes the test argument then, depending on this result, one computes *exactly one* of the other two arguments. As defined here, polygraphs algebraically describe the computation steps, but not the evaluation strategy. We leave such a task for further research.

Definition 96. Let $\mathcal{P}$ be a polygraphic program. For a 1-path u, a *value of type* u is a 2-path in $\langle \mathcal{P}_2^C \rangle$ with source $*$ and target u; their set is denoted by $[\![u]\!]$. Given a 2-path $f : u \Rightarrow v$, one denotes by $[\![f]\!]$ the (partial) map from $[\![u]\!]$ to $[\![v]\!]$ defined as follows : if t is a value of type u and if $t \star_1 f$ has a unique normal form t' that is a value (of type v), then $[\![f]\!](t)$ is t' ; otherwise f is undefined on t.

7.3 Polygraphic interpretations

For polygraphs, interpretations can be divided in two parts : the current, which corresponds more or less to a monotone interpretation, and the heat which provides termination. We present now the mathematical structure of currents and heat.

Interpretations are chosen over the positive natural numbers, denoted by $\mathbf{N}^*$. As shown in the previous section, natural numbers could be safely replaced by real numbers. However, this choice would induce some additional and tedious technicalities.

Definition 97. Let $\mathcal{P}$ be a 3-polygraphic program. A *functorial assignment* of $\mathcal{P}$ is a pair $\varphi = (\varphi_1, \varphi_2)$ made of :

1. a map φ_1 sending every 1-cell u of size n to a non-empty part of $(\mathbf{N}^*)^n$;
2. a map φ_2 sending every 2-cell $f : u \Rightarrow v$ to a weakly monotone (WMon) map from $\varphi_1(u)$ to $\varphi_1(v)$.

Using the functorial relations, one can extend the interpretation to any 1-path and 2-path.
- if u is a degenerate 2-path, then $\varphi_2(u)$ is the identity of $\varphi_1(u)$;
- $\varphi_1(u \star_0 v) = \varphi_1(u) \times \varphi_1(v)$;
- $\varphi_2(f \star_0 g) = \varphi_2(f) \times \varphi_2(g)$;
- $\varphi_2(f \star_1 g) = \varphi_2(g) \circ \varphi_2(f)$.

One simply writes φ for both φ_1 and φ_2. Intuitively, for every 2-cell $\blacksquare$, the map $\varphi(\blacksquare)$ tells us how $\blacksquare$, seen as a circuit gate, transmits currents downwards.

Definition 98. A *functorial interpretation* of a 3-polygraphic program is a functorial assignment such that for every 3-cell α of $\mathcal{P}$, we have $\varphi(s_2\alpha) \geq \varphi(t_2\alpha)$.

Example 99. Let $\mathcal{P}$ be a polygraphic program with no constructor 2-cell and no function 2-cell. Then, given a non-empty part $\varphi(\xi)$ of $\mathbf{N}^*$ for every 1-cell ξ, the following values extend φ into a functorial interpretation of $\mathcal{P}$:

$$\varphi\left(\times_{\xi,\zeta}\right)(x,y) \;=\; (y,x) \qquad \text{and} \qquad \varphi\left(\blacktriangle_{\xi}\right)(x) \;=\; (x,x).$$

Let us note that every functorial interpretation φ must send the 0-cell $*$ to some single-element part of $\mathbf{N}^*$. Hence, it must assign each $\bullet_\xi$ to the only map from $\varphi(\xi)$ to $\varphi(*)$.

Example 100. The following values extend the ones of Example 99 into a functorial interpretation of the polygraphic program $\mathcal{D}$ of division :

$$\varphi(\mathtt{n}) = \mathbf{N}^*, \qquad \varphi(\circ) = 1, \qquad \varphi(\diamond)(x) = x+1,$$

$$\varphi(\triangledown)(x,y) = \varphi(\blacktriangledown)(x,y) = x.$$

Example 101. For the polygraphic program $\mathcal{F}$ of fusion sort, we extend the functorial interpretation of Example 99 with the following values, where $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ stand for the rounding functions, respectively by excess and by default :

$$\varphi(\mathtt{n}) = \{1\}, \qquad \varphi(\mathtt{l}) = 2\mathbb{N}+1, \qquad \varphi(\circledcirc) = \varphi(\circ) = 1, \qquad \varphi(\triangledown)(x,y) = x+y+1,$$

$$\varphi(\blacklozenge)(x) = x, \qquad \varphi(\blacktriangledown)(x,y) = x+y-1, \qquad \varphi(\blacktriangle)(2x+1) = \left(2\cdot\left\lceil\frac{x}{2}\right\rceil+1,\ 2\cdot\left\lfloor\frac{x}{2}\right\rfloor+1\right).$$

Example 102. Let $\mathcal{P}$ be a polygraphic program. One denotes by ν the functorial interpretation on the subpolygraph $\left\langle \mathcal{P}_2^C \right\rangle$ defined, for every 1-cell ξ, by $\nu(\xi) = \mathbf{N}^*$ and, for every constructor 2-cell $\triangledown$ with arity n, by :

$$\nu(\triangledown)(x_1,\dots,x_n) \;=\; x_1+\cdots+x_n+1.$$

One checks that $\nu(t) = ||t||$ holds for every value t with coarity 1. Thus, given values $t_1, \dots, t_n$ with coarity 1, the following equality holds in $\mathbb{N}^n$:

$$\nu(t_1 \star_0 \cdots \star_0 t_n) \;=\; \big(\, ||t_1||, \dots, ||t_n||\,\big).$$

Thus ν can be used algebraically to describe the size of arguments of a function.

Analogous to Proposition 26, the following holds.

Proposition 103. *Let φ be a compatible functorial interpretation of a polygraphic program. Then, for every 3-path F, the inequality $\varphi(s_2F) \geq \varphi(t_2F)$ holds.*

Démonstration. The proof is by induction. □

Definition 104. Let $\mathcal{M}$ be an ordered set $(M,\preceq)$ equipped with a commutative monoid structure $(+,0)$. Let $\mathcal{P}$ be a 3-polygraph and let φ be a functorial interpretation of $\mathcal{P}$. A *differential interpretation of $\mathcal{P}$ over φ into M* is a map ∂ that sends each 2-path $\square$ of $\mathcal{P}$ with 1-source u to a monotone map $\partial\square$ from $\varphi(u)$ to M, such that the following conditions, called *differential relations*, are satisfied :

- If u is degenerate then $\partial u = 0$.
- $\partial(f \star_0 g)(x,y) = \partial f(x) + \partial g(y)$.
- $\partial(f \star_1 g) = \partial f + \partial g \circ \varphi(f)$.

Given some 3-cell α, we say that the interpretation is compatible with α if $\partial(s_2\alpha) \geq \partial(t_2\alpha)$. It is strictly compatible if $\partial(s_2\alpha) > \partial(t_2\alpha)$.

Intuitively, given a 2-cell $\square$, the map $\partial\square$ tells us how much heat it produces, when seen as a circuit gate, depending on the intensities of incoming currents. In order to compute the heat produced by a 2-path, one determines the currents that its 2-cells propagate and, from those values, the heat each one produces ; then one sums up all these heats.

By default, $\mathcal{M}$ is the set of natural numbers, with their original ordering and the standard monoid structure build from addition and zero.

Example 105. The *trivial* functorial interpretation of a 3-polygraph $\mathcal{P}$ sends every 1-cell to some fixed number $n \in \mathbf{N}^*$ and every 2-cell to the constant function n. Now, let us fix a family X of 2-cells in $\mathcal{P}$. One can check that the map $||\cdot||_X$ is the differential interpretation of $\mathcal{P}$ over the trivial interpretation, sending a 2-cell $\square$ to 1 if it is in X and 0 otherwise.

Example 106. We consider the differential interpretation of the division polygraphic program $\mathcal{D}$, over the functorial interpretation given in Example 100 sending every constructor and structure 2-cell to zero and :

$$\partial(x,y) \;=\; y+1 \qquad \text{and} \qquad \partial(x,y) \;=\; xy+x,$$

Example 107. For the polygraphic program $\mathcal{F}$ of fusion sort, we consider the differential interpretation, over the functorial interpretation of Example 101, into $(\mathbb{N},+,0)$, sending every constructor and structure 2-cells to zero and :

$$\partial(2x+1) = 2x^2+1, \quad \partial(2x+1) = \lfloor x/2 \rfloor + 1,$$
$$\partial(2x+1, 2y+1) = \begin{cases} 1 & \text{if } xy = 0, \\ x+y & \text{otherwise.} \end{cases}$$

Proposition 108. *Let ∂ be a differential interpretation of a polygraphic program $\mathcal{P}$ over a functorial interpretation φ, compatible with every 3-cell α. Then, for every 3-path F, the inequality $\partial(s_2F) \geq \partial(t_2F)$ holds. When the interpretation is strictly compatible with every 3-cell, then $\partial(s_2F) > \partial(t_2F)$.*

Definition 109. Let $\mathcal{P}$ be a 3-polygraph equipped with a functorial interpretation φ. One denotes by μ_φ the function sending every 2-cell with valence (m,n), *i.e.* with arity m and coarity n, to the following map from $\varphi(s_1)$ to $\mathbb{N}$:

$$\mu_\varphi(x_1,\ldots,x_m) \;=\; \max\{x_1,\ldots,x_m,y_1,\ldots,y_n\}$$

where $(y_1,\ldots,y_n)$ denotes $\varphi()(x_1,\ldots,x_m)$. Actually, μ_φ can be extended to a differential interpretation (but on the monoid $(\mathbf{N},\max,0)$) by means of the equations :

$$\begin{array}{ll} \mu_\varphi(u) = 0 & \text{if } u \text{ is degenerate} \\ \mu_\varphi(f \star_0 g)(x,y) = \max(\mu_\varphi f(x), \mu_\varphi g(y)) & \\ \mu_\varphi(f \star_1 g)(x) = \max(\mu_\varphi f(x), \mu_\varphi g \circ \varphi(f)(x)) & \end{array}$$

One says that φ is *conservative on* α when μ_φ is compatible with α. One says that φ is *conservative* when it is conservative on every 3-cell of $\mathcal{P}$.

Example 110. The functorial interpretations of Examples 100 and 101 are conservative. Indeed, we shall see later that their values on structure and constructor 2-cells ensure that they are conservative on structure 3-paths. Let us check conservativeness on, for example, the last computation 3-cell of the sort function 2-cell :

$$\begin{aligned} \mu_\varphi\left(\;\right)(1,1,2x+1) \;&=\; \max\{1,\;2x+1,\;2x+2,\;2x+3\} \\ &=\; 2x+3 \\ &=\; \max\{1,\;2x+1,\;2\cdot\lfloor x/2\rfloor+1,\;2\cdot\lceil x/2\rceil+1, \\ &\qquad 2\cdot\lfloor x/2\rfloor+2,\;2\cdot\lceil x/2\rceil+2,\;2x+3\} \\ &=\; \mu_\varphi\left(\;\right)(1,1,2x+1). \end{aligned}$$

When a functorial interpretation is both compatible and conservative, the intensities of currents inside 2-paths do not increase during computations. Analogous to Proposition 28, the following holds.

Proposition 111. *Let φ be a compatible and conservative functorial interpretation of a polygraphic program. Then, for every 3-path F, the inequality $\mu_\varphi(s_2F) \geq \mu_\varphi(t_2F)$ holds.*

Definition 112. A *polygraphic interpretation* of a 3-polygraph $\mathcal{P}$ is a pair (φ, ∂) made of a functorial interpretation φ of $\mathcal{P}$, together with a differential interpretation ∂ of $\mathcal{P}$ over φ and into $(\mathbb{N}, +, 0)$. φ is called the functorial part and ∂ the *differential part* of (φ, ∂).

Let us fix a 3-cell α. A polygraphic interpretation (φ, ∂) is *compatible (with α)* when both φ and ∂ are. It is *strictly compatible (with α)* when φ is compatible with α and ∂ is strictly compatible (with α). It is *conservative (on α)* when φ is.

Example 113. The functorial and differential interpretations we have built on the programs of division and of fusion sort are two examples of polygraphic interpretations that are conservative, compatible with every structure 3-cell and strictly compatible with every computation 3-cell.

We recall the following theorem :

Theorem 114 ([71])**.** *If a 3-polygraph has a polygraphic interpretation which is strictly compatible with all of its 3-cells, then it terminates.*

Proposition 115. *Let $\mathcal{P}$ be a 3-polygraph and let X be a set of 3-cells of $\mathcal{P}$. Let us assume that there exists a compatible polygraphic interpretation on $\mathcal{P}$ whose restriction to X is strictly compatible. Then $\mathcal{P}$ terminates if and only if $\mathcal{P} - X$ does.*

Example 116. Let us consider the polygraphic programs for division and fusion sort, given in Examples 93 and 94. We have seen that each one admits a compatible polygraphic interpretation that is strictly compatible with their computation 3-cells. Furthermore, as proved later, the structure 3-cells, alone, terminates. Thus Proposition 115 gives the termination of both polygraphic programs.

Actually, we produce below a standard differential interpretation that is strictly compatible with structure 3-cells. However, in general, it is not compatible, even in a non-strict way, with computation 3-cells : informally, each application of such a cell can increase the "structure heat". The purpose of the next proposition is to bound this potential augmentation.

Proposition 117. *Let $\mathcal{P}$ be a 3-polygraph, let α be a 3-cell of $\mathcal{P}$ and let F be an elementary 3-path in $\langle\alpha\rangle$. One assumes that $\mathcal{P}$ is equipped with a polygraphic interpretation (φ, ∂) such that φ is compatible with and conservative on α. Then, for every $x \in \varphi(s_1F)$, the following inequality holds :*

$$\partial(t_2F)(x) - \partial(s_2F)(x) \;\leq\; \sum_{\blacksquare \in \mathcal{P}_2} ||t_2(\alpha)||_{\blacksquare} \cdot \partial\blacksquare\left(\, \mu_\varphi(s_2F)(x),\; \ldots,\; \mu_\varphi(s_2F)(x)\,\right).$$

7.4 Complexity of polygraphic programs

In this section, we specialize polygraphic interpretations to polygraphic programs to get information on their complexity. We introduce additive polygraphic interpretations as we did for terms. By doing so, we get bounds on the size of computations, with respect to the size of the arguments. We conclude this work with a characterization of PTIME.

Definition 118. Let $\mathcal{P}$ be a polygraphic program. One says that a functorial interpretation φ of $\mathcal{P}$ is *additive* when, for every constructor 2-cell $\triangledown$ of arity n, there exists a non-zero natural number $c_{\triangledown}$ such that, for every $(x_1, \ldots, x_n)$ in $\varphi(s_1 \triangledown)$, the following equality holds in $\mathbb{N}$:

$$\varphi(\triangledown)(x_1, \ldots, x_n) \;=\; x_1 + \cdots + x_n + c_{\triangledown}.$$

In that case, one denotes by γ the largest of these numbers, *i.e.* :

$$\gamma \;=\; \max \left\{ c_{\triangledown},\; \triangledown \in \mathcal{P}_2^C \right\}.$$

A polygraphic interpretation is *additive* when its functorial part is.

Example 119. The functorial interpretations we have built for the polygraphic programs $\mathcal{D}$ and $\mathcal{F}$ are additive. In both cases, γ is 1.

Analogous to Proposition 32-(2,3), the following Proposition holds.

Proposition 120. *Let φ be an additive functorial interpretation of a polygraphic program $\mathcal{P}$. Then, for every value t, one has $\nu(t) \leq \varphi(t) \leq \gamma\nu(t)$, where ν is the functorial interpretation introduced in Example 102.*

Let $\blacksquare$ be a function 2-cell with arity m in a polygraphic program $\mathcal{P}$, equipped with an additive functorial interpretation φ. Thereafter, we denote by $M_{\blacksquare}$ the map from $\mathbb{N}^m$ to $\mathbb{N}$ defined by :

$$M_{\blacksquare}(x_1, \ldots, x_m) \;=\; \mu_{\varphi}\blacksquare\,\left(\, \gamma x_1,\; \ldots,\; \gamma x_m \,\right).$$

The next result uses the map $M_{\blacksquare}$ and the size of the initial arguments to bound the size of intermediate values produced during computations, hence of the arguments of potential recursive calls.

Proposition 121. *Let $\mathcal{P}$ be a polygraphic program, equipped with an additive, compatible and conservative functorial interpretation φ. Let $\blacksquare$ be a function 2-cell and let t be a value of type $s_1\blacksquare$. Then, for every 3-path F with source $t \star_1 \blacksquare$, the following inequality holds in $\mathbb{N}$:*

$$\mu_{\varphi}(t_2 F) \;\leq\; M_{\blacksquare} \circ \nu(t).$$

Example 122. Applied to Example 94, Proposition 121 tells us that, given a list t, any intermediate value produced by the computation of the sorted list $\blacklozenge(t)$ has its size bounded by $M_{\blacklozenge}(||t||) = ||t||$. This means that recursive calls made during this computation are applied to arguments of size at most $||t||$.

7.5 Cartesian polygraphic interpretations and the size of structure computations

Here we bound the number of structure 3-cells that can appear in a computation. Actually, we prove that structure 3-paths have polynomial size w.r.t. their input. To do that, we provide a generic interpretation, next called structure differential interpretation. So, the size of computation 3-cells can be computed modulo structure 3-path. In other words, for computation 3-cells, one may use the following interpretation.

Definition 123. Let $\mathcal{P}$ be a polygraphic program. A functorial interpretation φ of $\mathcal{P}$ is said to be *cartesian* when the following conditions hold, for every 1-cells ξ and ζ :

$$\varphi\left(\blacktriangle_{\xi}\right)(x) \;=\; (x,x) \qquad \text{and} \qquad \varphi\left(\times_{\xi,\zeta}\right)(x,y) \;=\; (y,x).$$

A polygraphic interpretation is *cartesian* when its functorial part is cartesian and when its differential part sends every constructor and structure 2-cell to zero.

Proposition 124. *If a functorial interpretation of a polygraphic program $\mathcal{P}$ is cartesian, then it is compatible with and conservative on all the structure 3-cells.*

So, for the "programmer", structure 2-cells are transparent as they were for term rewriting. We prove now that the structural computation do not last too long.

Definition 125. Let φ be a functorial interpretation of a polygraphic program $\mathcal{P}$. We denote by μ^S_φ and call *structure differential interpretation generated by* φ the differential interpretation of $\mathcal{P}$, over φ, that sends every constructor and function 2-cell to zero and such that the following hold :

$$\mu^S_\varphi \times (x,y) \;=\; xy, \qquad \mu^S_\varphi \blacktriangle (x) \;=\; x^2, \qquad \mu^S_\varphi \bullet (x) \;=\; x.$$

Lemma 126. *Let φ be a functorial interpretation of a polygraphic program $\mathcal{P}$. If φ is both additive and cartesian, then μ^S_φ is strictly compatible with all the structure 3-cells of $\mathcal{P}$.*

Definition 127. Let $\mathcal{P}$ be a polygraphic program. One denotes by K the maximum number of structure 2-cells one finds in the targets of computation 3-cells :

$$K = \max\left\{ ||t_2(\alpha)||_{\mathcal{P}^S_2} \,, \alpha \in \mathcal{P}^R_3 \right\}.$$

Let φ be an additive functorial interpretation of $\mathcal{P}$. For every function 2-cell $\blacksquare$ with arity m, one defines $S_{\blacksquare}$ as the map from $\mathbb{N}^m$ to $\mathbb{N}$ given by :

$$S_{\blacksquare}(x_1,\ldots,x_m) \;=\; K \cdot M^2_{\blacksquare}(x_1,\ldots,x_m).$$

The following lemma proves that, during a computation, if one applies a computation 3-cell, then the structure heat increase is bounded by a polynomial in the size of the arguments. Since structure computations have size bounded by the structure heat of the source, structure computations have polynomial size.

Lemma 128. *Let $\mathcal{P}$ be a polygraphic program, equipped with an additive, cartesian, compatible and conservative functorial interpretation φ. Let $\blacksquare$ be a function 2-cell and t be a value of type $s_1(\blacksquare)$. Let f and g be 2-paths such that $t \star_1 \blacksquare$ reduces into f which, in turn, reduces into g by application of a computation 3-cell α. Then, the following inequality holds in $\mathbb{Z}$:*

$$\mu^S_\varphi g - \mu^S_\varphi f \;\leq\; S_{\blacksquare} \circ \nu(t).$$

Example 129. For the polygraphic program of Example 94, we have $K = 1$. The polynomials bounding the structure interpretation increase after application of one of the computation 3-cells of this polygraphic program are :

$$S_{\bullet}(x) = x^2, \qquad S_{\blacktriangle}(x) = x^2, \qquad S_{\blacktriangledown}(x,y) = (x+y-1)^2.$$

7.6 The size of computations

It remains to make structure cells and computation cells work together.

Definition 130. Let $\mathcal{P}$ be a polygraphic program, with an additive polygraphic interpretation (φ, ∂). For every function 2-cell $\blacksquare$ with arity m, one denotes by $P_{\blacksquare}$ and by $Q_{\blacksquare}$ the maps from $\mathbb{N}^m$ to $\mathbb{N}$ defined by :

$$\begin{aligned} P_{\blacksquare}(x_1, \ldots, x_m) &= \partial\blacksquare\,\big(\gamma x_1, \ldots, \gamma x_m\big), \\ Q_{\blacksquare}(x_1, \ldots, x_m) &= P_{\blacksquare}(x_1, \ldots, x_m) \cdot \Big(1 + S_{\blacksquare}(x_1, \ldots, x_m)\Big). \end{aligned}$$

The following result bounds the number of computation 3-cells in a reduction 3-path, with respect to the size of the arguments.

Proposition 131. *Let $\mathcal{P}$ be a polygraphic program, equipped with an additive and cartesian polygraphic interpretation (φ, ∂) which is strictly compatible with every computation 3-cell. Let $\blacksquare$ be a function 2-cell and t be a value of type $s_1(\blacksquare)$. Then, for every 3-path F with source $t \star_1 \blacksquare$, the following inequality holds :*

$$|||F|||_{\mathcal{P}_3^R} \leq P_{\blacksquare} \circ \nu(t).$$

Proposition 132. *Let $\mathcal{P}$ be a polygraphic program, equipped with an additive and cartesian polygraphic interpretation (φ, ∂) which is strictly compatible with and conservative on every computation 3-cells. Let $\blacksquare$ be a function 2-cell and let t be a value of type $s_1\blacksquare$. Then, for every 3-path F with source $t \star_1 \blacksquare$, the following inequality holds :*

$$|||F||| \leq Q_{\blacksquare} \circ \nu(t).$$

Example 133. Let us compute these bounding maps for the fusion sort function 2-cell $\blacklozenge$ of the polygraphic program $\mathcal{F}$:

$$P_{\blacklozenge}(2x+1) = 2x^2 + 1 \qquad \text{and} \qquad Q_{\blacklozenge}(2x+1) = (2x^2+1) \cdot \big(1 + (2x+1)^2\big).$$

Let us fix a list $[i_1; \ldots; i_n]$ of natural numbers. One can check that, in $\mathcal{F}$, this list is represented by a 2-path t such that $\varphi(t) = ||t|| = 2n + 1$. The polynomial $P_{\blacklozenge}$ tells us that, during the computation of the sorted list $[\![\blacklozenge]\!](t)$, there will be at most $2n^2 + 1$ applications of computation 3-cells. The polynomial $Q_{\blacklozenge}$ bounds the total number of 3-cells of any type.

For example, when n is 2, one computes $[\![\blacklozenge]\!](t)$ by building a 3-path of size at most $Q_{\blacklozenge}(5) = 234$, containing no more than $P_{\blacklozenge}(5) = 9$ computation 3-cells.

7.7 Polygraphic programs and polynomial-time functions

Definition 134. Let $\mathcal{P}$ be a polygraphic program. A differential interpretation ∂ of $\mathcal{P}$ is *polynomial* when, for every function 2-cell $\blacksquare$, the map $\partial\blacksquare$ is bounded by a polynomial. A functorial interpretation φ of $\mathcal{P}$ is *polynomial* when μ_φ is. A polygraphic interpretation is *polynomial* when both its functorial part and differential part are.

We denote by **P** the set of polygraphic programs which are confluent, complete and which admit an additive, cartesian and polynomial polygraphic interpretation that is conservative on and strictly compatible with their computation 3-cells.

As a consequence of Proposition 132, programs in **P** can be simulated by a polynomial time procedure. Actually, the simulation can be done in both ways. Let us compute the polynomials with polygraphic programs.

Definition 135. Let us denote by $\mathcal{N}$ the polygraphic program with the following cells :

1. It has one 1-cell `n`.
2. Its 2-cells are the three possible structure 2-cells plus :
   (a) Constructor 2-cells : for zero and for the successor.
   (b) Function 2-cells : for addition and for multiplication.
3. Its 3-cells are the eight structure 3-cells plus the following computation 3-cells :

$$\Rrightarrow \qquad \Rrightarrow \qquad \Rrightarrow \qquad \Rrightarrow$$

Proposition 136. *The polygraphic program $\mathcal{N}$ is in* **P** *and it computes the addition and multiplication of natural numbers.*

Remark 137. So $\mathcal{N}$ computes addition and multiplication of natural numbers. As we have seen, it also computes duplication and permutation on them. As a consequence, for every polynomial P in $\mathbb{N}[x]$, one can choose a 2-path P in $\mathcal{N}$ such that $[\![P]\!]$ is P. Moreover, by induction, one proves that $\varphi(P) = P$ and that ∂P is bounded by a polynomial in $\mathbb{N}[x]$. Thus,

Theorem 138. *The polygraphic programs of* **P** *compute exactly the* Ptime *functions.*

Bibliographie

[1] Colin Adams, *The knot book*, American Mathematical Society, 2004.

[2] Beatriz Alarcón and Salvador Lucas, *Termination of innermost context-sensitive rewriting using dependency pairs*, FroCoS '07 : Proceedings of the 6th international symposium on Frontiers of Combining Systems (Berlin, Heidelberg), Springer-Verlag, 2007, pp. 73–87.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla, *Closed-Form Upper Bounds in Static Cost Analysis*, Journal of Automated Reasoning (2010), To appear.

[4] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini, *Resource usage analysis and its application to resource certification*, Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures (Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, eds.), Lecture Notes in Computer Science, vol. 5705, Springer, 2009, pp. 258–288.

[5] Roberto Amadio, *Max-plus quasi-interpretations*, Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings (Martin Hofmann, ed.), Lecture Notes in Computer Science, vol. 2701, Springer, 2003, pp. 31–45.

[6] ______, *Synthesis of max-plus quasi-interpretations*, Fundam. Inf. **65** (2004), no. 1-2, 29–60.

[7] Roberto Amadio, Sophie Coupet-Grimal, dal Zilio Silvano, and Line Jakubiec, *A functional scenario for bytecode verification of resource bounds.*, Computer Science Logic, 18th International Workshop, CSL 13th Annual Conference of the EACSL, Karpacz, Poland (Jerzy Marcinkowski and Andrzej Tarlecki, eds.), Lecture Notes in Computer Science, vol. 3210, Springer, 2004, pp. 265–279.

[8] Nils Andersen and Neil D. Jones, *Generalizing cook's transformation to imperative stack programs*, Proceedings of the Colloquium in Honor of Arto Salomaa on Results and Trends in Theoretical Computer Science (London, UK), Springer-Verlag, 1994, pp. 1–18.

[9] Oana Andrei and Hélène Kirchner, *A Higher-Order Graph Calculus for Autonomic Computing*, Graph Theory, Computational Intelligence and Thought. A Conference Celebrating Martin Charles Golumbic's 60th Birthday (Haifa Israël), 2008 (Anglais).

[10] Thomas Arts and Jürgen Giesl, *Proving innermost normalisation automatically*, RTA '97 : Proceedings of the 8th International Conference on Rewriting Techniques and Applications (London, UK), Springer-Verlag, 1997, pp. 157–171.

[11] ______, *Termination of term rewriting using dependency pairs*, Theoretical Computer Science **236** (2000), no. 1-2, 133–178.

[12] Martin Avanzini and Georg Moser, *Complexity analysis by rewriting*, Proc. 9th FLOPS, LNCS, vol. 4989, 2008, pp. 130–146.

[13] ———, *Complexity Analysis by Graph Rewriting*, Functional and Logic Programming (2010), 257–271.

[14] John Baez and Aaron Lauda, *A history of n-categorical physics*, draft version, 2006.

[15] Patrick Baillot, Ugo Dal Lago, and Jean-Yves Moyen, *On quasi-interpretations, blind abstractions and implicit complexity*, CoRR **abs/cs/0608030** (2006).

[16] Patrick Baillot and Kazushige Terui, *Light types for polynomial time computation in lambda-calculus*, Proceedings of the 19th Symposium on Logic in Computer Science (LICS 04), 2004, pp. 266–275.

[17] ———, *Light types for polynomial time computation in lambda calculus*, Information and Computation **207** (2009), no. 1, 41–62.

[18] Jean-Pierre Banâtre, Pascal Fradet, and Yves Radenac, *A generalized higher-order chemical computation model*, ENTCS, vol. 135, 2006.

[19] Saugata Basu, Richard Pollack, and Marie.-Françoise Roy, *Algorithms in real algebraic geometry*, Springer, Berlin Heidelberg New York, 2003.

[20] Stephen Bellantoni and Stephen Cook, *A new recursion-theoretic characterization of the poly-time functions*, Computational Complexity **2** (1992), 97–110.

[21] Stephen Bellantoni and Isabel Oitavem, *Separating NC along the δ axis*, Theoretical Computer Science **318** (2004), 57–78.

[22] Ahmed Ben Cherifa and Pierre Lescanne, *Termination of rewriting systems by polynomial interpretations and its implementation*, Science of Computer Programming **9** (1987), no. 2, 137–160.

[23] Gerard Berry and Gerard Boudol, *The chemical abstract machine*, POPL '90 : Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM, 1990, pp. 81–94.

[24] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio, *The computability path ordering : the end of a quest*, Computer Science Logic (Bertinoro) (Springer, ed.), Lecture Notes in Computer Science, vol. 5213, 2008.

[25] Manuel Blum, *A machine-independent theory of the complexity of recursive functions*, J. ACM **14** (1967), no. 2, 322–336.

[26] Guillaume Bonfante, *Constructions d'ordres, analyse de la complexité*, Thèse, Institut National Polytechnique de Lorraine, 2000.

[27] ———, *Some programming languages for LOGSPACE and PTIME*, 11th International Conference on Algebraic Methodology and Software Technology - AMAST'06 (Kuresaare/Estonie), 07 2006.

[28] ———, *Observation of implicit complexity by non confluence*, Electronic Proceedings in Theoretical Computer Science **23** (2010).

[29] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet, *Complexity classes and rewrite systems with polynomial interpretation*, Proceedings of the 12th International Workshop on Computer Science Logic (London, UK), Springer-Verlag, 1999, pp. 372–384.

[30] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet, *Algorithms with polynomial interpretation termination proof*, J. Funct. Program. **11** (2001), no. 1, 33–53.

[31] Guillaume Bonfante and Florian Deloup, *Complexity invariance of real interpretations*, Theory and Applications of Models of Computation, 7th Annual Conference, TAMC 2010 (Jan Kratochvíl, Angsheng Li, Jirí Fiala, and Petr Kolman, eds.), LNCS, vol. 6108, Springer, 2010, pp. 139–150.

[32] Guillaume Bonfante, Bruno Guillaume, Mathieu Morey, and Guy Perrier, *Réécriture de graphes de dépendances pour l'interface syntaxe-sémantique*, July 2010.

[33] Guillaume Bonfante and Yves Guiraud, *Polygraphic programs and polynomial-time functions*, Logical Methods in Computer Science (LMCS) **5** (2009), 1–37.

[34] Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem, *Towards an implicit characterization of NC^k*, Computer Science Logic '06 (Zoltán Èsik, ed.), Lecture Notes in Computer Science, vol. 4207, Springer, 2006, pp. 212–224.

[35] Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem, *Recursion Schemata for NCk*, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings (Bertinoro Italie) (Michael Kaminski and Simone Martini, eds.), vol. 5213, Springer, 2008, pp. 49–63.

[36] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen, *On lexicographic termination ordering with space bound certifications*, Perspectives of System Informatics, PSI 2001, Novosibirsk, Russia (Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, eds.), Lecture Notes in Computer Science, vol. 2244, Springer, Jul 2001.

[37] ———, *Quasi-Interpretations and Small Space Bounds*, Rewrite Techniques and Applications (J. Giesl, ed.), Lecture Notes in Computer Science, vol. 3467, Springer, April 2005, pp. 150–164.

[38] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen, *Quasi-interpretation : a way to control ressources*, Theoretical Computer Science (2009), 35 (Anglais), accepted for publication.

[39] Guillaume Bonfante, Jean-Yves Marion, Jean-Yves Moyen, and Romain Péchoux, *Synthesis of Quasi-interpretation*, Proceedings of LCC'05, June 2005.

[40] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux, *A characterization of alternating log time by first order functional programs*, LPAR (Springer, ed.), Lecture Notes in Computer Science, vol. 4246, 2006, pp. 90–104.

[41] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux, *Quasi-interpretation Synthesis by Decomposition : An application to higher-order programs*, ICTAC (Macao Chine), 2007.

[42] Albert Burroni, *Higher-dimensional word problems with applications to equational logic*, Theoretical Computer Science **115** (1993), no. 1, 43–62.

[43] Ashok Chandra, Dexter Kozen, and Larry Stockmeyer, *Alternation*, Journal of the ACM (JACM) **28** (1981), no. 1, 114–133.

[44] Adam Cichon and Jean-Yves Marion, *The light lexicographic path ordering*, Tech. report, Loria, 2000, Workshop Rule.

[45] Horatiu Cirstea and Claude Kirchner, *The rewriting calculus — Part I*, Logic Journal of the Interest Group in Pure and Applied Logics **9** (2001), no. 3, 427–463.

[46] ———, *The rewriting calculus — Part II*, Logic Journal of the Interest Group in Pure and Applied Logics **9** (2001), no. 3, 465–498.

[47] A. Cobham, *The intrinsic computational difficulty of functions*, Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science (Y. Bar-Hillel, ed.), North-Holland, Amsterdam, 1962, pp. 24–30.

[48] Loïc Colson, *About primitive recursive algorithms*, Theor. Comput. Sci. **83** (1991), no. 1, 57–69.

[49] Stephen Cook, *Characterizations of pushdown machines in terms of time-bounded computers*, Journal of the ACM **18** (1971), no. 1, 4–18.

[50] Michel Coste, Henri Lombardi, and Marie-Françoise Roy, *Dynamical method in algebra : Effective nullstellensätze*, Annals of Pure and Applied Logic **111** (2001), 203–256.

[51] Patrick Cousot, *Abstract interpretation*, ACM Comput. Surv. **28** (1996), no. 2, 324–328.

[52] Ugo dal Lago and Simone Martini, *Derivational complexity is an invariant cost model*, Foundational and Practical Aspects of Resource Analysis (FOPARA '09), 2009.

[53] Olivier Danvy, *An analytical approach to programs as data objects*, BRICS, University of Aarhus, 2006, Doctor Scientarum degree in Computer Science.

[54] Nachum Dershowitz, *Orderings for term-rewriting systems*, Theoretical Computer Science **17** (1982), no. 3, 279–301.

[55] ______, *Termination of rewriting*, Journal of Symbolic Computation (1987), 69–115.

[56] Nachum Dershowitz and Yuri Gurevich, *A natural axiomatization of computability and proof of church's thesis*, Bulletin of Symbolic Logic **14** (2008), no. 3, 299–350.

[57] Nachum Dershowitz and Jean-Pierre Jouannaud, *Handbook of theoretical computer science vol.b*, ch. Rewrite systems, pp. 243–320, 1990.

[58] M. V. H. Fairtlough and S. S. Wainer, *Ordinal complexity of recursive definitions*, Inf. Comput. **99** (1992), no. 2, 123–153.

[59] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca, *A logical account of pspace*, SIGPLAN Not. **43** (2008), no. 1, 121–131.

[60] ______, *Soft linear logic and polynomial complexity classes*, Electron. Notes Theor. Comput. Sci. **205** (2008), 67–87.

[61] Marco Gaboardi and Brian Redmond, *Towards a light logic for pspace*, Logic and Complexity, LCC, 2010.

[62] Jürgen Giesl, *Generating polynomial orderings for termination proofs*, Proc. 6th RTA, Lecture Notes in Computer Science, vol. 914, 1995, pp. 426–431.

[63] Jürgen Giesl and Art Middeldorp, *Innermost termination of context-sensitive rewriting*, Tech. report, In Proceedings of the 6th International Conference on Developments in Language Theory (DLT 2002). Lecture Notes in Computer Science, 2002.

[64] Jean-Yves Girard, *Linear logic*, Theoretical Computer Science **50** (1987), 1–102.

[65] ______, *Light linear logic*, Information and Computation **143** (1998), no. 2, 175–204.

[66] Isabelle Gnaedig and Hélène Kirchner, *Termination of rewriting under strategies*, ACM Trans. Comput. Log. **10** (2009), no. 2.

[67] Erich Grädel and Yuri Gurevich, *Tailoring recursion for complexity*, Journal of symbolic logic **60** (1995), no. 3, 952–69.

[68] Etienne Grandjean, *Linear time algorithms and np-complete problems*, SIAM J. Comput. **23** (1994), no. 3, 573–597.

[69] Etienne Grandjean and Thomas Schwentick, *Machine-independent characterizations and complete problems for deterministic linear time*, SIAM J. Comput. **32** (2003), no. 1, 196–230.

[70] Yves Guiraud, *Prèsentations d'opèrades et systèmes de rèècriture*, Ph.D. thesis, UniversitÈ Montpellier 2, June 2004.

[71] ______, *Termination orders for 3-dimensional rewriting*, Journal of Pure and Applied Algebra **207** (2006), no. 2, 341–371.

[72] ______, *The three dimensions of proofs*, Annals of Pure and Applied Logic **141** (2006), no. 1-2, 266–295.

[73] ______, *Two polygraphic presentations of Petri nets*, Theoretical Computer Science **360** (2006), no. 1-3, 124–146.

[74] Sumit Gulwani, *The art of invariant generation for symbolic loop bound analysis*, Invited Talk at CAV, 2009.

[75] Yuri Gurevich, *Algebras of feasible functions*, SFCS '83 : Proceedings of the 24th Annual Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 1983, pp. 210–214.

[76] ______, *The sequential ASM thesis*, EATCS Bulletin **67** (1999), 93–124.

[77] Nao Hirokawa and Georg Moser, *Automated complexity analysis based on the dependency pair method*, IJCAR (Alessandro Armando, Peter Baumgartner, and Gilles Dowek, eds.), Lecture Notes in Computer Science, vol. 5195, Springer, 2008, pp. 364–379.

[78] Dieter Hofbauer, *Termination proofs by context-dependent interpretations*, Proceedings of the 12th International Conference on Rewriting Techniques and Applications, RTA-01 (Utrecht, The Netherlands) (Aart Middeldorp, ed.), Lecture Notes in Computer Science, vol. 2051, Springer-Verlag, 2001, pp. 108–121.

[79] ______, *Proving termination with matrix interpretations*, Proceedings of the 17th International Conference on Rewriting Techniques and Applications, RTA-06 (Seattle, USA), Lecture Notes in Computer Science, vol. 4098, Springer-Verlag, 2006, pp. 328–342.

[80] Martin Hofmann, *Linear types and non-size-increasing polynomial time computation*, Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99), 1999, pp. 464–473.

[81] ______, *The strength of non-size increasing computation*, POPL '02 : Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, New York, NY, USA, 2002, pp. 260–269.

[82] Martin Hofmann and Ulrich Schöpp, *Pure pointer programs with iteration*, CSL '08 : Proceedings of the 22nd international workshop on Computer Science Logic (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 79–93.

[83] Hoon Hong and Dalibor Jakus, *Testing positiveness of polynomials*, J. Autom. Reasoning **21** (1998), no. 1, 23–38.

[84] Gérard Huet, *Confluent reductions : Abstract properties and applications to term rewriting systems*, Journal of the ACM **27** (1980), no. 4, 797–821.

[85] Neil Immerman, *Relational queries computable in polynomial time*, Inf. Control **68** (1986), no. 1-3, 86–104.

[86] Herman Ruge Jervell, *Ordering finite labeled trees*, Computability In Europe (A Beckmann, C Dimitracopoulos, and B Löwe, eds.), Lecture Notes in Computer Science, vol. 5028, 2008.

[87] Neil Jones, *Constant time factors do matter*, Symposium on Theory of Computing Proceedings, ACM, 1993, pp. 602–611.

[88] ———, *Computability and complexity, from a programming perspective*, MIT Press, 1997.

[89] ———, *Logspace and ptime characterized by programming languages*, Theroretical Computer Science **228** (1999), 151–174.

[90] Neil D. Jones and Lars Kristiansen, *A flow calculus of mwp-bounds for complexity analysis*, ACM Trans. Comput. Logic **10** (2009), no. 4, 1–41.

[91] Samuel Kamin and Jean-Jacques Lévy, *Attempts for generalising the recursive path orderings.*, Tech. report, Univerity of Illinois, Urbana, 1980, Unpublished note. Accessible on http ://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html.

[92] D.E. Knuth and P.B. Bendix, *Simple word problems in universal algebras*, Computational problems in abstract algebra (Pergamon) (J. Leech, ed.), 1970.

[93] Mark W. Krentel, *The complexity of optimization problems*, Journal of computer and system sciences **36** (1988), 490–519.

[94] Mukkai S. Krishnamoorthy and Paliath Narendran, *On recursive path ordering*, Theoretical Computer Science **40** (1985), no. 2-3, 323–328.

[95] Lars Kristiansen and Paul J. Voda, *Complexity classes and fragments of c*, Inf. Process. Lett. **88** (2003), no. 5, 213–218.

[96] Yves Lafont, *Towards an algebraic theory of boolean circuits*, Journal of Pure and Applied Algebra **184** (2003), no. 2-3, 257–310.

[97] ———, *Soft linear logic and polynomial time*, Theoretical Computer Science **318** (2004), 163–180.

[98] D.S. Lankford, *On proving term rewriting systems are noetherien*, Tech. report, 1979.

[99] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram, *The size-change principle for program termination*, Symposium on Principles of Programming Languages, vol. 28, ACM press, january 2001, pp. 81–92.

[100] Daniel Leivant, *A foundational delineation of computational feasiblity*, Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91), 1991.

[101] ———, *A characterization of nc by tree recurrence.*, 39th Annual Symposium on Foundations of Computer Science, FOCS'98, 1998, pp. 716–724.

[102] Daniel Leivant and Jean-Yves Marion, *Predicative functional recurrence and poly-space*, TAPSOFT'97, Theory and Practice of Software Development (M. Bidoit and M. Dauchet, eds.), Lecture Notes in Computer Science, vol. 1214, Springer, Apr 1997, pp. 369–380.

[103] ———, *A characterization of alternating log time by ramified recurrence*, Theoretical Computer Science **236** (2000), no. 1-2, 192–208.

[104] Jean-Louis Loday, *Generalized bialgebras and triples of operads*, preprint, 2006.

[105] Salvador Lucas, *On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting*, Appl. Algebra Eng., Commun. Comput. **17** (2006), no. 1, 49–73.

[106] ———, *Practical use of polynomials over the reals in proofs of termination*, PPDP '07 : Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming (New York, NY, USA), ACM, 2007, pp. 39–50.

[107] Jean-Yves Marion and Jean-Yves Moyen, *Efficient first order functional program interpreter with time bound certifications*, Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France (Michel Parigot and Andrei Voronkov, eds.), Lecture Notes in Computer Science, vol. 1955, Springer, Nov 2000, pp. 25–42.

[108] ______, *Termination and resource analysis of assembly programs by petri nets*, Tech. report, LORIA - INRIA, 2003.

[109] Jean-Yves Marion and Romain Péchoux, *Resource analysis by sup-interpretation*, FLOPS, Lecture Notes in Computer Science, vol. 3945, Springer-Verlag, 2006, pp. 163–176.

[110] ______, *A characterization of nck*, TAMC, Lecture Notes in Computer Science, vol. 4978, Springer, 2008.

[111] Jean-Yves Marion and Romain Péchoux, *Characterizations of polynomial complexity classes with a better intensionality*, PPDP (Sergio Antoy and Elvira Albert, eds.), ACM, 2008, pp. 79–88.

[112] Robin Milner, *The space and motion of communicating agents*, Cambridge University Press, 2009, to appear.

[113] Virgile Mogbil and Vincent Rahli, *Uniform circuits, & Boolean proof nets*, Lecture Notes in Computer Science LNCS 4514, Logical Foundations of Computer Science 2007 International Symposium of Logical Foundations of Computer Science, LFCS 2007 (New York États-Unis d'Amérique) (Anil Nerode Sergei N. Artemov, ed.), LNCS 4514, Springer, 2007, pp. 401–421.

[114] Yiannis Moschovakis, *What is an algorithm*, Mathematics Unlimited – 2001 and Beyond (Björn Engquist and Wilfried Schmid, eds.), Springer, 2001, pp. 919–936.

[115] Georg Moser, *Proof theory at work : Complexity analysis of term rewrite systems*, University of Innsbruck, 2009, Habilitation Thesis.

[116] Georg Moser and Andreas Schnabl, *Proving quadratic derivational complexities using context dependent interpretations*, Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 5117, Springer-Verlag, 2008, pp. 276–290.

[117] Georg Moser, Andreas Schnabl, and Johannes Waldmann, *Complexity analysis of term rewriting based on matrix and context dependent interpretations*, IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008) (Dagstuhl, Germany) (Ramesh Hariharan, Madhavan Mukund, and V Vinay, eds.), Leibniz International Proceedings in Informatics, vol. 2, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008.

[118] Jean-Yves Moyen, *Resource control graphs*, ACM Trans. Comput. Logic **10** (2009), no. 4, 1–44.

[119] Jean-Yves Moyen and Metnani Ryma, *Mwp matrices and quasi-interpretations*, personal communication.

[120] Friedrich Neurauter and Aart Middeldorp, *Polynomial interpretations over the reals do not subsume polynomial interpretations over the integers*, Proceedings of the 21st International Conference on Rewriting Techniques and Applications (Christopher Lynch, ed.), Leibniz International Proceedings in Informatics (LIPIcs), vol. 6, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 243–258.

[121] Max Newman, *On theories with a combinatorial definition of "equivalence"*, Annals of Math. **43** (1942), no. 2, 223–243.

[122] Karl-Heinz Niggl, *The μ-measure as a tool for classifying computational complexity*, Archive for Mathematical Logic (2000), to appear.

[123] Karl-Heinz Niggl and Henning Wunderlich, *Certifying polynomial time and linear/polynomial space for imperative programs*, SIAM J. Comput. **35** (2006), no. 5, 1122–1147.

[124] Gheorghe Paun, *Membrane computing : An introduction*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[125] Rósza Péter, *Rekursive funktionen*, Akadémiai Kiadó, Budapest, 1966, English translation : *Recursive Functions*, Academic Press, New York, 1967.

[126] D. Plaisted, *A recursively defined ordering for proving termination of term rewriting systems*, Tech. Report R-78-943, Department of Computer Science, University of Illinois, 1978.

[127] Femke van Raamsdonk, *On termination of higher-order rewriting*, RTA '01 : Proceedings of the 12th International Conference on Rewriting Techniques and Applications (London, UK), Springer-Verlag, 2001, pp. 261–275.

[128] H.E. Rose, *Subrecursion*, Oxford university press, 1984.

[129] Vladimir Yu. Sazonov, *Polynomial computability and recursivity in finite domains*, Elektronische Informationsverarbeitung und Kybernetik **16** (1980), no. 7, 319–323.

[130] Ulrich Schöpp, *Stratified bounded affine logic for logarithmic space*, LICS '07 : Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (Washington, DC, USA), IEEE Computer Society, 2007, pp. 411–420.

[131] Olha Shkaravska, Marko van Eekelen, and Ron van Kesteren, *Polynomial size analysis of first-order shapely functions*, CoRR **abs/0902.2073** (2009).

[132] H. Simmons, *Derivation and computation*, Tracts in theoretical computer science, vol. 51, Cambridge, 2000.

[133] Joachim Steinbach, *Generating polynomial orderings*, Information Processing Letters **49** (1994), 85–93.

[134] Gilbert Stengle, *A nullstellensatz and a positivstellensatz in semialgebraic geometry*, Mathematische Annalen **207** (1973), no. 2, 87–97.

[135] Alfred Tarski, *A decision method for elementary algebra and geometry*, University of California Press, 1951, 2nd edition.

[136] Marc van den Brand, Pierre-Etienne Moreau, and Christophe Ringeissen, *The elan environment : a rewriting logic environment based on asf+sdf technology -system demonstration-*, Proceedings : 2nd International Workshop on Language Descriptions, Tools and Applications, volume 65. Electronic Notes in Theoretical Computer Science, 2002.

[137] Moshe Y. Vardi, *The complexity of relational query languages (extended abstract)*, STOC '82 : Proceedings of the fourteenth annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 1982, pp. 137–146.