# Analyse de la complexité des programmes par interprétation sémantique

Romain Péchoux

▶ To cite this version:

Romain Péchoux. Analyse de la complexité des programmes par interprétation sémantique. Autre [cs.OH]. Institut National Polytechnique de Lorraine, 2007. Français. NNT : 2007INPL084N . tel-01752904v1

HAL Id: tel-01752904
https://hal.univ-lorraine.fr/tel-01752904v1

Submitted on 29 Mar 2018 (v1), last revised 16 Sep 2008 (v2)

**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.

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Nancy-Université
Institut National Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

# Analyse de la complexité des programmes par interprétation sémantique

## THÈSE

présentée et soutenue publiquement le 14 novembre 2007

pour l'obtention du

**Doctorat de l'Institut National Polytechnique de Lorraine**
**(spécialité informatique)**

par

Romain Péchoux

**Composition du jury**

| | | |
|---|---|---|
| *Rapporteurs :* | Roberto Amadio | Professeur, Université Paris VII |
| | Neil Jones | Professeur, DIKU, University of Copenhagen |
| *Examinateurs :* | Patrick Baillot | Chargé de Recherche, CNRS |
| | Claude Kirchner | Directeur de Recherche, INRIA |
| | Jean-Yves Marion | Professeur, École des Mines de Nancy, INPL |
| | Simona Ronchi Della Rocca | Professeur, Università di Torino |
| | Paul Zimmermann | Directeur de Recherche, INRIA |
| *Invité :* | Guillaume Bonfante | Chargé de Recherche, INRIA |

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

Loria

# Remerciements

J'aimerais remercier de nombreuses personnes pour avoir contribué, à titres divers, à l'élaboration de ce document. J'adresse donc des remerciements les plus chaleureux et sincères :

- À Clémence, mon épouse, pour m'avoir soutenu pendant ces trois années de thèse.
- À mon directeur de thèse, Jean-Yves Marion pour m'avoir encouragé et pour avoir su me donner de bonnes pistes et intuitions de recherche.
- À Roberto Amadio et à Neil Jones qui ont accepté d'être les rapporteurs de mon manuscrit.
- Aux autres membres de mon jury de thèse, Patrick Baillot, Claude Kirchner, Luigi Liquori, Simona Ronchi Della Rocca et Paul Zimmermann.
- À tous les membres de ma famille, mes parents et ma soeur.
- Aux membres de l'équipe CARTE avec lesquels j'ai pu échanger des idées, Guillaume Bonfante, Olivier Bournez, Johanne Cohen et Isabelle Gnaedig.
- Aux membres du projet ACI CRISS avec lesquels j'ai travaillé entre 2004 et 2006.
- Aux différents collègues de bureau, que j'ai côtoyés pendant cette thèse et qui ont toujours contribué à maintenir une ambiance chaleureuse (et studieuse) sur notre lieu de travail. Dans l'ordre chronologique, Jean-Yves Moyen, Matthieu Kaczmarek, Marco Gaboardi, Emmanuel Hainry, Daniel Graça, Anne Bonfante et Octave Boussaton, ainsi qu'à tous les collègues de l'équipe PROTHEO, Germain, Florent, Antoine, Colin et à Sébastien de l'équipe PAROLE.
- À tous mes autres amis dont la bande de l'École des Mines, Charles ($\times$ 2), Blandine, Émeline, Fred, Domi, Fabrice et Nadim, et le gang des dentistes, Soph, Jul, Séb, Nat, Gillou, Pat et Julie.
- À mes collègues enseignants, dont Azim Roussanaly, Antoine Tabbone et Odile Thiery, qui m'ont encadré et conseillé au cours des trois années de monitorat que j'ai effectuées à l'UFR de Mathématiques et d'Informatique de l'Université de Nancy 2.

# Table des matières

**Remerciements** i

**Introduction** 3

**I Les quasi-interprétations** 11

I.1 Définitions 14
- I.1.1 Syntaxe des programmes fonctionnels au premier ordre 14
- I.1.2 Sémantique des programmes fonctionnels au premier ordre 15
- I.1.3 Précédence 17
- I.1.4 Arbre des appels 17
- I.1.5 Assignations partielles 18
- I.1.6 Quasi-interprétations 19

I.2 Propriétés des quasi-interprétations 22
- I.2.1 Machines de Turing et classes de complexité polynomiales 22
- I.2.2 Lemme fondamental 23
- I.2.3 Ordres récursifs sur les chemins 26
- I.2.4 Une caractérisation de FPtime 28
- I.2.5 Une caractérisation de FPspace 31

I.3 Synthèse de quasi-interprétations 34
- I.3.1 Définition 35
- I.3.2 Décidabilité du problème de la synthèse sur **Max-Poly** $\{\mathbb{R}^+\}$ 37
- I.3.3 Synthèse de quasi-interprétations **Max-Plus** 41

I.4 Modularité des quasi-interprétations 49
- I.4.1 Union disjointe 50
- I.4.2 Union à constructeurs partagés 52
- I.4.3 Union hiérarchique 56

I.5 Application à d'autres langages de programmation 67
- I.5.1 Application à la vérification de bytecode 67
- I.5.2 Application à des systèmes de threads concurrents et interactifs 68
- I.5.3 Application à des programmes d'ordre supérieur 69

I.6 Conclusion 72

**II Les sup-interprétations** 73

II.1 Définitions 76
- II.1.1 Adjonction d'opérateurs 76
- II.1.2 Fraternités 77
- II.1.3 Assignations partielles 78

II.1.4 Sup-interprétations . . . 79
II.1.5 Poids . . . 81
II.2 Critères permettant le contrôle des ressources en espace . . . 82
II.2.1 Critère quasi-amical . . . 82
II.2.2 Critère pour les programmes non-terminants . . . 90
II.2.3 Critère pour les algorithmes de type « diviser pour régner » . . . 96
II.3 Application aux travaux précédents . . . 100
II.3.1 Comparaison avec les quasi-interprétations . . . 100
II.3.2 Application aux paires de dépendance . . . 103
II.3.3 Application au size-change principle . . . 107
II.4 Une caractérisation des classes de complexité $NC^k$ . . . 110
II.4.1 Rappels sur $ALogTime$, $NC^K$ et $NC$ . . . 110
II.4.2 Programmes arborescents et sympathiques . . . 114
II.4.3 Caractérisations de $ALogTime$, $NC^K$ et $NC$ . . . 119
II.5 Synthèse de sup-interprétations . . . 130
II.5.1 Synthèse avec propriété sous-terme . . . 130
II.5.2 Synthèse sans propriété sous-terme . . . 130
II.6 Conclusion . . . 132

**III Extension des sup-interprétations à un langage orienté objet 135**
III.1 Programmation orientée objet . . . 137
III.1.1 Syntaxe des programmes . . . 137
III.1.2 Sémantique . . . 140
III.2 Sup-interprétations et poids . . . 142
III.2.1 Assignations . . . 142
III.2.2 Sup-interprétations . . . 144
III.2.3 Poids . . . 147
III.3 Critère fraternel . . . 148
III.3.1 Définition . . . 148
III.3.2 Propriétés des programmes fraternels . . . 150
III.4 Synthèse de sup-interprétations . . . 153
III.4.1 Poids d'une méthode . . . 153
III.4.2 Méthodes fraternelles . . . 153
III.5 Caractérisations de classes de complexité . . . 155
III.5.1 Une caractérisation de FPSPACE . . . 155
III.5.2 Une caractérisation de FPTIME . . . 157
III.6 Conclusion . . . 159

**Conclusion 161**

**Bibliographie 163**

**Index 175**

# Introduction

> Je n'ai pas besoin de m'occuper de ce que je ferai plus tard. Je devais faire ce que j'ai fait. Je n'ai pas besoin de découvrir quelles choses je découvrirai plus tard. Dans la nouvelle science, chaque chose vient à son tour, telle est son excellence.
>
> *(Lautréamont, Poésies II)*

Le présent ouvrage étudie l'analyse de la complexité des programmes par interprétations sémantiques ; c'est-à-dire le contrôle des ressources utilisées par des programmes à l'aide d'un outil spécifique, les interprétations sémantiques. Dans cette introduction, nous cherchons à motiver l'objet d'une telle étude en partant de la notion très générale de « besoins humains », puisque le contrôle des ressources s'inscrit dans une volonté de satisfaire efficacement un besoin. Nous présenterons la notion de ressources informatiques qui permet de satisfaire tous les besoins humains liés à l'utilisation de l'outil informatique. Nous introduirons ensuite la théorie de la complexité qui permet la mise en place d'une étude formelle des ressources informatiques. Puis, nous plongerons un peu plus dans les détails en dressant un éventail des différentes méthodes permettant de contrôler les ressources d'un programme. Enfin, nous présenterons notre démarche sur l'étude des ressources à l'aide d'interprétations sémantiques en donnant une trame narrative à laquelle le lecteur pourra se référer.

**Les besoins humains :**

La notion de besoin résulte de l'interaction entre un individu et son environnement. Les sociologues ont pris l'habitude de classifier les besoins humains en deux catégories principales :

– Les besoins primaires ou physiologiques qui représentent les besoins indispensables à la survie de l'Homme comme la respiration ou encore la reproduction de l'espèce.
– Les besoins secondaires ou matériels qui rassemblent tous les besoins qui ne sont pas indispensables à la survie de l'Homme comme la programmation ou la recherche du minimum d'une fonction.

Cette classification reste cependant très subjective, et, est donc sujette à controverses puisqu'elle dépend fortement du contexte économique et social des individus. En caricaturant légèrement, on peut considérer que l'accès à un ordinateur est en passe de devenir un besoin primaire dans les sociétés occidentales tandis qu'il représente un besoin secon-

daire pour les indiens Pataxo ou les pygmées Mbuti.

Le psychologue américain Abraham Maslow a affiné cette hiérarchie des besoins dans un diagramme pyramidal présenté dans la figure suivante :

Pyramide des besoins de Maslow

D'après lui, l'Homme cherche à satisfaire prioritairement les besoins d'un niveau inférieur de la pyramide avant de s'intéresser aux niveaux supérieurs. De bas en haut de la pyramide, on trouve :

- Les besoins physiologiques qui correspondent au maintien des fonctions vitales appelé aussi homéostasie. Claude Bernard la définit comme étant « l' équilibre dynamique qui nous maintient en vie ».
- Le besoin de sécurité qui correspond à la nécessité pour l'individu d'assurer son intégrité physique et son intégrité morale.
- Le besoin d'appartenance qui s'inscrit dans la volonté de l'individu de communiquer et d'appartenir à un groupe social.
- Les besoins d'estime qui correspondent à une nécessité d'être respecté, de se respecter et de respecter les autres. On peut constater que l'estime des autres l'emporte en général sur l'estime de soi.
- Enfin, le dernier besoin est un besoin d'auto-réalisation qui se traduit par un goût de l'apprentissage et la volonté de participer à des activités totalement désintéressées.

Les besoins qui se cachent derrière l'usage de l'outil informatique sont, pour la plupart, des besoins d'appartenance et de sécurité. Ils correspondent globalement aux utilisations personnelle et professionnelle de l'informatique. Le développement des technologies de la communication et de l'information et en particulier d'Internet, à la fin du siècle dernier, est révélateur d'une tentative de satisfaction du besoin d'appartenance. Les internautes ont pu communiquer par mail ou partager leurs opinions sur les forums et autres sites de chat. Derrière leurs buts premiers, les blogs et les communautés de développement de logiciels (on peut penser à sourceforge.net) ou de jeux en ligne cachent une réelle volonté d'afficher son appartenance à un groupe social. Le besoin de sécurité en Informatique est, quant à lui, intimement lié à la mise en application des outils informatiques à l'échelle industrielle. Par exemple, l'utilisateur veut éviter qu'un logiciel embarqué bogue par manque d'espace mémoire. Il veut éviter toute intrusion d'un utilisateur anonyme dans une base de données confidentielle ou, encore, il veut crypter un message à envoyer afin

de ne pas être espionné. Ce besoin de sécurité est en passe de devenir un enjeu majeur dans nos sociétés puisqu'il s'étend désormais aux utilisations personnelle et familiale de l'ordinateur : Aujourd'hui, la grande majorité des ordinateurs vendus dans le commerce est équipée d'un antivirus qui permet de se prémunir d'éventuelles attaques du monde extérieur.

**Les ressources informatiques :**

La notion de besoin est étroitement liée à celle de ressource. En économie, la ressource correspond aux « moyens matériels dont dispose un pays, une région, une collectivité ». Ainsi, les ressources représentent les moyens qui vont permettre la satisfaction du besoin. Cependant, la notion de ressource est plus restrictive que celle de besoin puisqu'elle dénote en plus la finitude des moyens dont on dispose. Car les ressources sont limitées, et l'Homme se doit de les exploiter convenablement pour satisfaire ses besoins et ainsi produire de la valeur. En informatique, les ressources représentent « l'ensemble des moyens dont dispose un ordinateur pour exécuter un ou plusieurs programmes ». Les ressources informatiques sont diverses et dépendent en général du matériel et de ses limites physiques. L'informatique théorique a modélisé l'ensemble de ces contraintes physiques comme des ressources de temps et d'espace. Ces deux ressources schématisent toutes les restrictions matérielles à l'échelle humaine. La ressource en temps va correspondre à l'exécution en temps fini d'un programme. Par exemple, un utilisateur va vouloir que son programme termine en lui retournant un résultat en un temps plus ou moins raisonnable. Cependant, elle n'est pas toujours pertinente. Par exemple, un système d'exploitation va s'exécuter en continu après que l'ordinateur a été mis sous tension et il serait dommageable qu'il s'arrête au bout d'un temps donné puisqu'il doit prendre en charge le bon fonctionnement de tous les autres programmes. De ce point de vue là, les ressources en espace sont beaucoup plus intéressantes que les ressources en temps puisque leur pertinence ne dépend pas du type de programme considéré. L'ordinateur dispose de supports de mémorisation, comme le disque dur et la mémoire vive, et leur capacité de stockage est physiquement limitée. Ainsi il sera nécessaire de contrôler la mémoire de certaines applications afin d'éviter des bogues ou des attaques. Par exemple, si l'on intègre un logiciel embarqué dans une automobile, quelle garantie possède-t-on sur le fait que le logiciel ne dépassera pas la quantité mémoire maximale disponible lors de son exécution. De telles problématiques sont primordiales dans des domaines à forts enjeux stratégiques et financiers, comme l'aérospatial et l'armement, mais elles tendent aussi à le devenir dans notre quotidien. Que faire si ma voiture ne répond plus ou si mon téléphone portable bogue après le téléchargement et l'exécution d'un jeu Java ?

**La théorie de la complexité :**

La théorie de la complexité a été introduite afin de proposer un cadre formel permettant de répondre à ce type de questions. Elle correspond à l'étude de la difficulté des problèmes informatiques en utilisant une notion de mesure de complexité [Blu67] et se doit de :

- définir clairement cette notion de mesure de complexité qui va permettre de quantifier les ressources à maîtriser.
- permettre le calcul de la complexité d'un programme relativement à la mesure de complexité en considération.
- étudier la complexité des programmes indépendamment du modèle de calcul en considération ; c'est-à-dire indépendamment de la machine utilisée pour effectuer les calculs.

Car la richesse de l'informatique est intimement liée à la diversité des machines et des modèles de calcul à disposition. En général, la théorie de la complexité utilise les machines de Turing [Tur36] comme modèle fondamental. Les ouvrages suivants [Jon97, Pap94] comparent d'autres modèles aux machines de Turing tout en donnant une vision très détaillée de la théorie de la complexité. Les machines de Turing demeurent néanmoins le modèle de calcul le plus populaire pour des raisons scientifiques et historiques et seront décrites en détail en section I.2. En substance, elles sont constituées de un ou plusieurs rubans[1] permettant de mémoriser les symboles d'un alphabet donné dans différentes cases mémoire et d'un état permettant de modéliser l'état courant de la machine. Une étape du calcul correspond à un changement d'état courant et à l'écriture d'un nouveau symbole, parfois identique, dans une case de chaque ruban. La complexité en temps d'un calcul est défini comme le nombre d'écritures d'un symbole durant un calcul tandis que la complexité en espace correspond au nombre maximal de cases mémoire utilisé pendant le calcul. Il est d'usage d'utiliser différentes classes de fonctions, par exemple l'ensemble des fonctions polynomiales ou l'ensemble des fonctions exponentielles, afin de hiérarchiser la complexité de différents problèmes dans ce que l'on nomme classe de complexité. Par problème, on sous-entend problème de décision ; c'est-à-dire une question mathématique que l'on peut résoudre en répondant par l'affirmative ou par la négative. L'ensemble des problèmes pouvant être résolus par une Machine de Turing possédant une complexité en temps bornée par un polynôme en les entrées est appelé PTIME tandis que l'ensemble des problèmes pouvant être résolus par une Machine de Turing possédant une complexité en espace bornée par un polynôme en les entrées est appelé PSPACE. La diversité des classes de complexité étudiées dans la littérature est impressionnante. Elle dépend des différentes notions de mesure de complexité établies et de la multitude de modèles de calcul existants. Le calcul quantique est un exemple de modèle de calcul plus ésotérique que la simple Machine de Turing. Par ailleurs, les machines de Turing peuvent être étendues au non-déterminisme ou à des modèles randomisés. Le non-déterminisme correspond à la possibilité pour la machine d'effectuer un choix à chaque étape du calcul. La machine calcule un problème de manière non-déterministe si au moins une suite de choix permet de résoudre le problème. On définit de manière analogue les classes NPTIME et NPSPACE pour les machines de Turing non-déterministes. Ces différentes classes de problèmes permettent l'obtention d'une hiérarchie de problèmes :

$$\text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} = \text{NPSPACE}$$

[1]Le calcul d'une machine à plusieurs rubans diffère beaucoup du calcul effectué par une machine à un seul ruban.

Cette hiérarchie peut évidemment être affinée afin d'obtenir des classes de complexité plus petites ou plus grandes [Pap94], par exemple, en utilisant des machines de Turing alternantes. Elle permet aussi de dresser un comparatif entre le pouvoir de calcul de différents modèles de calcul. Des problèmes particulièrement intéressants de la théorie de la complexité, appelés problèmes de séparation, consistent à déterminer si, pour deux classes de complexité $A$ et $B$ données, on peut démontrer que $A \subset B$. Un tel résultat démontre que les problèmes de la classe $A$ sont strictement « plus faciles » que les problèmes de la classe $B$. De nombreux problèmes de séparation sont encore ouverts comme le fameux PTIME $=^?$ NPTIME pour ne citer que lui. Les problèmes de décision étant relativement limités, on étend la notion de classe de complexité à la notion de problème en général (ou de fonctions), définissant de la sorte des classes de complexité telles que FPTIME, l'ensemble des fonctions calculables par par une Machine de Turing possédant une complexité en temps bornée par un polynôme en les entrées. On a l'inclusion évidente suivante PTIME $\subseteq$ FPTIME puisque tout problème de décision est une fonction particulière. Étant donné un modèle de calcul raisonnable, on appelle caractérisation d'une classe de complexité $A$ une restriction d'un modèle de calcul de sorte qu'une fonction est calculée dans $A$ si et seulement si elle est calculée par la restriction du modèle de calcul. Parmi les plus célèbres, citons les travaux de Cobham [Cob62] qui caractérisent FPTIME à l'aide d'une algèbre de fonctions. On dit alors que la caractérisation est machine-indépendante puisqu'elle ne dépend pas d'un modèle de calcul mais d'une algèbre de fonctions. Soient $F$ un ensemble de fonctions et $O$ un ensemble d'opérations, une algèbre de fonctions se définit comme étant le plus petit ensemble contenant les fonctions initiales de $F$ et stable par les opérations de $O$. On la note $[F; O]$. Cobham a démontré que l'algèbre de fonctions $\left[\mathbf{0}, \mathbf{S_0}, \mathbf{S_1}, \pi^k_j, \#; \mathtt{COMP}, \mathtt{BRN}\right]$ caractérise exactement FPTIME, où $\mathbf{0} : \mathbb{N} \to \mathbb{N}$ est la fonction constante vérifiant $\forall y \in \mathbb{N},\ \mathbf{0}(y) = 0$, $\mathbf{S}_i : \mathbb{N} \to \mathbb{N}$ est la fonction successeur vérifiant $\forall y \in \mathbb{N},\ \mathbf{S}_i(y) = 2 \times y + i$, $\pi^k_j$ est la $j$-ième projection d'un $k$-uplet vérifiant $\pi^k_j(y_1, ..., y_k) = y_j$ et $\#$ est la fonction vérifiant $\forall x, y \in \mathbb{N},\ x\#y = 2^{|x|\times|y|}$ et où les opérations COMP et BRN sont définies respectivement par :

$$
\begin{aligned}
\mathtt{COMP}(\mathtt{g}, \mathtt{h})(y_1, \cdots, y_k) &= \mathtt{g}(\mathtt{h}_1(y_1, \cdots, y_k), \ldots, \mathtt{h}_n(y_1, \cdots, y_k)) \\
\mathtt{BRN}(\mathtt{g}_0, \mathtt{g}_1, \mathtt{h}_0, \mathtt{h}_1, k)(0, y) &= \mathtt{g}_0(y) \\
\mathtt{BRN}(\mathtt{g}_0, \mathtt{g}_1, \mathtt{h}_0, \mathtt{h}_1, k)(1, y) &= \mathtt{g}_1(y) \\
\mathtt{BRN}(\mathtt{g}_0, \mathtt{g}_1, \mathtt{h}_0, \mathtt{h}_1, k)(\mathbf{S_0}(x), y) &= \mathtt{h}_0(x, y, \mathtt{BRN}(\mathtt{g}_0, \mathtt{g}_1, \mathtt{h}_0, \mathtt{h}_1, k)(x, y)) \\
\mathtt{BRN}(\mathtt{g}_0, \mathtt{g}_1, \mathtt{h}_0, \mathtt{h}_1, k)(\mathbf{S_1}(x), y) &= \mathtt{h}_1(x, y, \mathtt{BRN}(\mathtt{g}_0, \mathtt{g}_1, \mathtt{h}_0, \mathtt{h}_1, k)(x, y)) \\
\text{avec } \mathtt{BRN}(\mathtt{g}_0, \mathtt{g}_1, \mathtt{h}_0, \mathtt{h}_1, k)(x, y) &\leq k(x, y)
\end{aligned}
$$

**Études de la complexité des programmes :**

Ce précédent résultat peut être vu comme la pierre angulaire de nos travaux. S'il a le mérite de définir une base solide pour l'étude de la complexité des programmes, il limite en pratique l'étude des programmes puisqu'il requiert la connaissance préalable d'une borne supérieure sur les ressources utilisées, en l'occurrence la borne $k$ dans le schéma de

récurrence `BRN`. En d'autres termes, on désire que l'analyse effectuée soit décidable afin que l'on puisse l'automatiser. Une esquisse de réponse à cet inconvénient est apportée par les différents types d'analyse statique de la complexité des programmes. L'analyse statique de la complexité d'un programme permet d'étudier un programme avant son exécution et de garantir des propriétés sur les ressources qu'il va utiliser pendant son exécution. Afin d'éviter tout recours à une borne supérieure comme dans les travaux présentés ci-dessus, des caractérisations syntactiques des classes de complexité appelées analyses prédicatives de la récurrence ont été proposées comme la safe recursion [BC92], comme dans le lambda-calcul [LM93] ou comme le data tiering [Lei94]. Le principe fondamental de ces caractérisations est de remplacer la borne supérieure sur les fonctions calculées par des schémas de récurrence plus restrictifs. Cette restriction établit une hiérarchie entre les fonctions et porte sur les arguments des appels récursifs qui ne peuvent plus être utilisés comme arguments de récurrence d'une fonction de niveau inférieur. De telles caractérisations sont très élégantes mais possèdent l'inconvénient d'avoir une faible intensionnalité, c'est-à-dire de caractériser un nombre assez faible d'algorithmes naturels. La notion d'intensionnalité sera détaillée plus amplement dans l'introduction du chapitre I. Les travaux suivants permettent de combler en partie cette lacune tout en restant décidables :

- Des études descendant de la Logique Linéaire de Girard [Gir87] fournissent différentes caractérisations de Ptime, par Girard et al. [Laf04, Gir98, GSS92], par Baillot et Mogbil [BM04] ou par Gaboardi et Ronchi Della Rocca [GRDR07], en utilisant des logiques ou des règles de typage restreintes.
- Dans [Hof00], Hofmann a introduit un type de ressource atomique et utilise un système de types linéaires. Leur combinaison permet de contrôler les ressources des programmes à l'ordre supérieur.
- Les travaux de Fagin [Fag73], Jones [Jon00], Goerdt [Goe92], Gurevich [Gur83] et Sazonov [Saz80] permettent de caractériser différentes classes de complexité et d'étudier la complexité des programmes à l'aide de modèles finis. Nous conseillons vivement au lecteur intéressé par ce domaine de recherche de lire l'ouvrage suivant [Imm99].
- Une autre approche concerne l'étude des langages impératifs soit à l'aide de restrictions sur les boucles [Nig98, Nig00, KN04] par Niggl et al., soit à l'aide de conditions restrictives sur une algèbre de matrices [NW06, KJ06].
- Enfin la dernière approche concerne l'étude des programmes à l'aide d'interprétations et, en particulier, de quasi-interprétations et de sup-interprétations. La notion de quasi-interprétation a été introduite par Bonfante [Bon00], Marion [Mar00, Mar03] et Marion-Moyen [MM00] et s'inspire de la notion d'interprétation polynomiale utilisée par Lankford [Lan79] pour prouver la terminaison de systèmes de réécriture. La notion de sup-interprétation est une généralisation de la notion de quasi-interprétation introduite dans [MP06b, MP07c] qui peut être étendue à des langages de programmation autres que les seuls langages fonctionnels.

**Contributions :**

L'usage de ces interprétations et, en particulier, des interprétations à valeur sur des polynômes pour contrôler les ressources en temps et en espace est l'objet principal de cette thèse. La suite de ce travail sera donc focalisée sur les différents résultats obtenus lors de l'étude de ces outils :

1. Le chapitre I traite de la notion de quasi-interprétation et fournit pour la première fois une vue d'ensemble des différents résultats obtenus dans ce domaine de recherche. La quasi-interprétation est une interprétation sémantique définie sur les programmes fonctionnels au premier ordre et qui associe une fonction à chaque symbole d'un programme. Ces fonctions se doivent de vérifier certaines conditions spécifiques, comme les propriétés sous-terme et de monotonie, afin d'obtenir une borne correcte sur les ressources utilisées et la quasi-interprétation doit vérifier des inégalités relatives au programme considéré. Après avoir redéfini la syntaxe et la sémantique des programmes considérés, la notion de quasi-interprétation est introduite dans la première section (I.1). La deuxième section (I.2) introduit les premiers résultats fondamentaux sur les quasi-interprétations. À savoir, des bornes sur la taille des valeurs calculées par un programme utilisant des quasi-interprétations ainsi que des caractérisations des ensembles de fonctions calculables en espace et en temps polynomiaux. La section I.3 traite d'un problème crucial abordé dans [BMMP05, Péc05] concernant les quasi-interprétations : la synthèse. Il consiste à déterminer si un programme admet ou non une quasi-interprétation et se révèle indispensable dans une perspective d'automatisation de cette étude. On démontre que ce problème est décidable sur des ensembles de fonctions donnés. La grande complexité de ce problème pour des classes restreintes de fonctions polynomiales nous pousse à étudier la modularité des quasi-interprétations dans la section I.4. La modularité consiste à découper un programme en sous-programmes afin de diminuer la complexité de la synthèse de quasi-interprétations. Ce problème a été traité dans [BMP07]. On démontre qu'en plus de diminuer la complexité de la synthèse, on arrive à étudier la complexité d'un plus grand nombre d'algorithmes. Enfin, dans la section I.5 de ce chapitre, nous mentionnons d'autres langages de programmation auxquels les quasi-interprétations ont été étendues.
2. Le chapitre II étudie la notion de sup-interprétation, introduite dans [MP06b, MP07c], qui généralise celle de quasi-interprétation. L'introduction des sup-interprétations a été motivée par une volonté de capturer un plus grand nombre d'algorithmes que ne le permettait l'usage des quasi-interprétations. La section II.1 définit brièvement les sup-interprétations en ajoutant au langage une notion d'opérateurs qui correspond à des opérations primitives du langages. La section II.2 introduit différents critères permettant d'analyser la complexité des programmes du premier ordre, dont, en particulier, le critère quasi-amical introduit dans [MP06a]. Il est démontré dans la section II.3 que la notion de sup-interprétation généra-

lise strictement la notion de quasi-interprétation. De plus, on montre qu'elle peut être appliquée efficacement à différents critères de terminaison comme les paires de dépendance ou le size-change principle. Enfin, on s'intéresse au pouvoir de caractérisation des sup-interprétations dans la section II.4 en généralisant des résultats de [BMP06]. L'outil étant plus performant que les quasi-interprétations, il permet d'étudier des classes de complexité plus petites que FPtime et FPspace comme les $NC^k$. Enfin, la section II.5 s'intéresse elle-aussi au problème de la synthèse pour les sup-interprétations. Ce problème est encore plus compliqué pour les sup-interprétations que pour les quasi-interprétations. Nonobstant cette difficulté, on y démontre que l'on peut obtenir des heuristiques efficaces, proposées dans [MP07b], permettant de synthétiser des sup-interprétations.

3. Pour finir, le chapitre III étend la notion de sup-interprétation à des langages orientés objet et impératifs [MP07a](si l'on restreint le langage objet en considération), justifiant ainsi la pertinence des notions d'interprétation considérées. Nous commençons par décrire la syntaxe et la sémantique du langage objet considéré en section III.1. Puis, nous étendons les sup-interprétations et toutes les notions qui en découlent à ce type de programme dans la section III.2. Dans la section III.3, nous définissons un critère permettant de contrôler les ressources des programmes orientés-objet à l'aide des sup-interprétations. Nous démontrons que les programmes vérifiant ce critère calculent des objets dont la taille reste bornée polynomialement en la taille des entrées. Puis, nous étendons ce critère aux méthodes dans la section III.4 de manière à obtenir des heuristiques pour synthétiser les sup-interprétations des méthodes. Enfin, dans la section III.5 nous montrons que les sup-interprétations permettent à nouveau de caractériser les classes de complexité polynomiales à l'aide des programmes orientés-objet.

# I Les quasi-interprétations

> L'intelligence [...] repose moins sur l'espoir, qui nous soutient dans les moments de détresse, que sur les pronostics plus sûrs qu'elle tire d'une juste estimation des ressources dont on dispose.
>
> *(Thucydide, La guerre du Péloponèse)*

Dans cette section, nous allons introduire la notion de quasi-interprétation [Mar03, Moy03, MM00, BMM01, BMM07] ainsi que toutes les propriétés et tous les résultats qui en découlent. Une quasi-interprétation est un outil permettant l'analyse statique de la complexité des programmes fonctionnels au premier ordre. Ce type d'étude revêt une importance particulière pour de nombreuses applications comme les technologies embarquées telles que les cartes à puces ou la téléphonie mobile et se révèle d'un intérêt prépondérant dans le domaine de la sécurité informatique. De nombreuses autres approches essaient de résoudre le même type de problématique comme le monitoring qui consiste à étudier l'évolution d'un système en temps réel. Cependant, le monitoring ne permet pas l'obtention de l'espace mémoire nécessaire à un programme avant son exécution. Une approche complémentaire repose sur une génération de tests. On obtient ainsi des bornes inférieures sur la mémoire requise par un programme, sans que jamais aucune indication ne soit donnée sur une éventuelle borne supérieure. Notre approche s'inscrit dans une visée différente. Nous cherchons à contrôler les ressources en fournissant et en vérifiant des certificats. Un certificat doit garantir que les ressources nécessaires à l'exécution d'un code compilé seront suffisantes. En d'autres termes, le programme doit être certifié « sans dépassement de mémoire » avant exécution.

Les travaux sur les quasi-interprétations s'inspirent, d'une part, de la notion d'interprétation polynomiale introduite dans [MN70, Lan79] afin de prouver la terminaison des programmes et, d'autre part, de précédents travaux sur la complexité implicite des programmes comme la safe recursion [BC92], le lambda-calcul [LM93] ou le data tiering [Lei94]. Cependant, les quasi-interprétations capturent plus d'algorithmes naturels que ces précédents travaux basés sur des schémas de récurrence restrictifs. On dit alors que les quasi-interprétations possèdent une plus grande intensionnalité.

L'intensionnalité est un concept de logique et de linguistique qui découle des travaux de Frege [Fre92] et qui s'oppose à l'extensionnalité. Frege distingue les notions de « sens » et de « dénotation ». La dénotation est l'objet désigné par un signe ou une expression, c'est-à-dire l'objet auquel on fait référence, tandis que le sens représente la manière dont

l'objet est spécifié. Le sens est donc le mode de déclaration de la dénotation. Ces deux notions sont reprises de manière relative dans la théorie de la philosophie du langage de Carnap [Car56] pour désigner l'extension et l'intension. Ainsi, si l'on considère l'expression « un informaticien », son intension est le concept d'informaticien ainsi que ses propriétés tandis que son extension est l'ensemble de tous les informaticiens. Ce genre d'analyse se généralise aux propositions. La proposition « Bill croit que son butineur est bogué » n'est pas équivalente à « Bill croit que son navigateur est bogué » pour peu que Bill ignore que les mots « butineur » et « navigateur » désignent le même objet. Ces propositions, de même que la majorité des propositions dans une langue naturelle, sont donc intensionnelles : leur valeur de vérité est fonction de leur dénotation mais aussi de l'interprétation que l'on en fait, et donc de leur sens. Ces notions se généralisent aisément aux langages de programmation en considérant que l'intension correspond à l'algorithme tandis que l'extension correspond à la fonction calculée par l'algorithme. Les études suivantes [Jon00, Mar00, Hof02, Nig98, Ben01], bien qu'elles ne le mentionnent pas toujours explicitement, ont largement contribué à bien distinguer les notions d'intension et d'extension en informatique. Par exemple, les algorithmes de tri par insertion, de tri par fusion et de tri rapide (Quicksort) présentés dans [CLR90] calculent tous la même fonction et sont donc extensionnellement équivalents. En revanche, les algorithmes leur correspondant diffèrent. Leur intension n'est donc pas la même.

La pertinence d'un outil d'analyse de la complexité d'un programme sera donc jugée à l'aune de sa complétude intensionnelle, c'est-à-dire de la quantité d'algorithmes « naturels » qu'il capture. De ce point de vue, la quasi-interprétation représente un outil très performant. En effet, comme nous l'avons fait remarquer au paragraphe précédent, l'intensionnalité dépend directement de l'interprétation que l'on fait d'une expression donnée. Il résulte de cette remarque que les quasi-interprétations sont intensionnellement plus puissantes que les approches sus-mentionnées. En particulier, des approches basées sur des schémas de récurrence sont toutes d'ordre syntaxique tandis que les quasi-interprétations combinent élégamment analyses syntaxique et sémantique.

L'article [BMM07] fournit une introduction détaillée aux quasi-interprétations. Une quasi-interprétation est une assignation d'une fonction monotone et ayant la propriété sous-terme (Cf. définition I.13) à l'ensemble des symboles apparaissant dans un programme. Ces assignations vérifient des systèmes d'inégalités qui fournissent une borne supérieure sur la taille des calculs effectués par les symboles d'un programme. En outre, la combinaison des quasi-interprétations et d'ordres de terminaison récursifs sur les chemins (Recursive Path Orderings ou RPO) [KL80, Der82] permet de caractériser différentes classes de complexité telles que la classe des fonctions calculables en temps polynomial ou, encore, la classe des fonctions calculables en espace polynomial. Les principales caractéristiques des quasi-interprétations sont les suivantes :

1. L'analyse de programme par quasi-interprétation capture une très grande classe d'algorithmes, dont les algorithmes qui ont une longueur de dérivation exponentielle mais dont la fonction dénotée peut être calculée en temps polynomial à l'aide de techniques de programmation dynamique [MM00].

2. La notion de quasi-interprétation est suffisamment souple pour pouvoir être éten-

due à d'autres langages. Il a été demontré dans [ACGDZJ04, DZG05] que la vérification des ressources s'étend au bytecode en commençant par compiler un programme fonctionnel du premier ordre qui admet une quasi-interprétation. Enfin, la notion de quasi-interprétation a aussi été étendue à la programmation réactive dans [ADZ04].

3. Dans [BMMP05, BMM07], il a été démontré que la synthèse de quasi-interprétation introduite dans [Ama05], qui consiste à trouver une quasi-interprétation pour un programme donné, est décidable en temps exponentiel pour peu que l'on considère des max-polynômes, c'est-à-dire une algèbre de polynômes munis de l'opération max, de degré borné sur les nombres réels. Ainsi, la notion de quasi-interprétation se révèle automatisable lorsque l'on considère que son codomaine est un ensemble de polynômes sur les réels. Le choix des nombres réels est d'autant plus intéressant qu'il existe des programmes n'admettant que des quasi-interprétations avec au moins un coefficient irrationnel. Enfin, lorsque l'on restreint notre étude à des petites classes de polynômes à valeur sur les réels, on peut démontrer que le problème de la synthèse est NP-complet [Ama05, BMMP05].

4. La modularité des quasi-interprétations a été étudiée dans [BMP07]. La problématique est de diviser un programme et de s'intéresser aux sous-programmes ainsi obtenus. Si les deux sous-programmes admettent une quasi-interprétation, on peut se demander quelles sont les propriétés que nous pouvons en déduire pour le programme initial. Par exemple, la quasi-interprétation est-elle modulaire : Le programme principal admet-il, lui aussi, une quasi-interprétation ? Il a été démontré que les quasi-interprétations sont modulaires dans le cas particulier d'une union disjointe lorsqu'aucun symbole n'est partagé par les deux sous-programmes. En revanche, les quasi-interprétations ne sont pas modulaires si on divise le programme de manière arbitraire. Cependant, dans le cas d'une union à constructeurs partagés (constructor-sharing), où seuls les symboles de constructeur sont partagés entre les deux sous-programmes, on peut tout de même en déduire une borne polynomiale. Et, dans le cas d'une union hiérarchique, lorsque certains symboles de constructeur d'un sous-programme sont des symboles de fonction de l'autre sous-programme, on peut en déduire une borne polynomiale sous certaines restrictions d'ordre syntaxique. De tels résultats augmentent l'intensionnalité de l'outil, puisque l'on peut désormais analyser des algorithmes pour lesquelles la méthode initiale des quasi-interprétation échouait. Ces algorithmes sont pour la plupart des algorithmes de codage et de décodage d'un alphabet vers un autre. Le principal intérêt d'un découpage en sous-programme réside donc dans la possibilité de diminuer la complexité de la synthèse de quasi-interprétations en utilisant des techniques dites « diviser pour régner ».

5. Enfin, il est possible d'appliquer les quasi-interprétations à des programmes d'ordre supérieur en utilisant des techniques de défonctionnalisation. Tout programme à l'ordre supérieur est spécialisé, dans un premier temps, en un programme au premier ordre extensionnellement équivalent, puis, dans un deuxième temps, on applique nos outils d'analyse au programme ainsi obtenu.

## I.1 Définitions

### I.1.1 Syntaxe des programmes fonctionnels au premier ordre

Nous décrivons ici la syntaxe d'un langage de programmation générique au premier ordre. Le vocabulaire $\Sigma = \langle Var, Cns, Fct \rangle$ se compose de trois ensembles disjoints de symboles qui représentent respectivement l'ensemble des variables, l'ensemble des symboles de constructeur et l'ensemble des symboles de fonction. L'arité d'un symbole est le nombre de ses arguments. Un programme **p** de notre langage se compose d'une séquence de définitions $def_1, \cdots, def_m$ qui décrivent des symboles de fonction et qui sont définies par la grammaire de la figure I.1. Nous utiliserons parfois la notation $\langle Var, Cns, Fct, \mathcal{R} \rangle$ où $\mathcal{R} = \{def_1, \cdots, def_m\}$ afin de désigner un tel programme **p**.

$$
\begin{array}{llll}
\texttt{Définition} \ni def & ::= & \texttt{f}(x_1, \cdots, x_n) = e^{\texttt{f}} \\
\texttt{Expression} \ni e & ::= & x \mid \mathbf{c}(e_1, \cdots, e_n) \mid \texttt{f}(e_1, \cdots, e_n) \\
 & & \mid \mathbf{Case}\ x_1, \cdots, x_n\ \mathbf{of}\ \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell \\
\texttt{Motif} \ni p & ::= & x \mid \mathbf{c}(p_1, \cdots, p_n)
\end{array}
$$

où $x, x_1, \ldots, x_n \in Var$ sont des variables, $\mathbf{c} \in Cns$ est un symbole de constructeur, $\texttt{f} \in Fct$ est un symbole de fonction et $\overline{p_i}$ est une séquence de $n$ motifs. La notation $\overline{e}$ désigne toute séquence d'expressions $e_1, \ldots, e_n$, lorsque $n$ est clairement déterminé par le contexte.

FIG. I.1: Syntaxe d'un programme **p**

L'opérateur **Case** est un symbole dont le rôle est de permettre le filtrage de motifs et dont l'usage est strictement limité à une position « outermost ». On ne trouve donc ce symbole qu'à la racine d'une expression. En d'autres termes, une expression $e^j$ ou $e_i$ ne contient aucun **Case** dans la grammaire de la figure I.1. Cette restriction n'est pas drastique puisqu'un programme ne la vérifiant pas, c'est-à-dire contenant un **Case** imbriqué, peut être transformé en temps linéaire en un programme de taille équivalente qui la satisfait.

Une variable du corps $e^{\texttt{f}}$ d'une définition est soit une variable apparaissant dans la liste des attributs $x_1, \cdots, x_n$ de la définition de $\texttt{f}$, lorsque l'opérateur **Case** n'est pas utilisé, soit une variable apparaissant dans un motif. Lorsque l'opérateur **Case** apparaît dans une expression, on suppose que les motifs ne se chevauchent pas et que les variables de ces motifs sont utilisées de manière linéaire. De telles restrictions garantissent que les programmes considérés sont confluents [Hue80]. Les programmes ainsi définis sont très proches des systèmes de réécriture étudiés dans [KK, BN98, DJ90, Klo92] et peuvent être transformés de manière linéaire en systèmes de réécriture en éliminant l'opérateur **Case**.

### I.1.2 Sémantique des programmes fonctionnels au premier ordre

Le domaine de calcul d'un programme $\mathbf{p}$ est l'ensemble $\mathcal{V}^* = \mathcal{V} \cup \{\mathbf{Err}\}$ où l'ensemble des valeurs $\mathcal{V}$ représente l'algèbre de termes constructeurs $\mathcal{T}(Cns)$, défini par $\mathcal{T}(Cns) \ni v ::= b \mid b(v_1, \cdots, v_n)$ avec $b \in Cns$, et $\mathbf{Err}$ est un symbole sémantique d'arité nulle retourné comme sortie par un programme lorsqu'une erreur advient, comme lorsqu'aucune définition ne peut être appliquée.

Une substitution $\sigma$ est une fonction partielle de $Var$ dans $\mathcal{V}$. L'application d'une substitution $\sigma$ à une expression $e$ est notée $e\sigma$. Étant données une substitution $\sigma$ et une expression $e$, la signification du jugement $e\sigma \downarrow w$ est la suivante : l'expression $e\sigma$ s'évalue en la valeur $w$ appartenant à $\mathcal{V}^*$. Si aucune définition n'est applicable, alors une erreur survient et $e\sigma \downarrow \mathbf{Err}$.

Notre langage possède une sémantique d'appel par valeur définie comme étant la clôture des règles de la figure I.2.

$$\frac{t_1 \downarrow w_1 \ldots t_n \downarrow w_n}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(w_1, \cdots, w_n)} \ \mathbf{c} \in Cns \text{ et } \forall i, w_i \neq \mathbf{Err}$$

$$\frac{e \downarrow u \quad \exists \sigma, i \ : \ p_i\sigma = u \quad e_i\sigma \downarrow w}{\mathbf{Case}\ \overline{x}\ \mathbf{of}\ \overline{p_1} \to e_1 \ldots \overline{p_\ell} \to e_\ell \downarrow w} \ \mathbf{Case} \text{ et } u \neq \mathbf{Err}$$

$$\frac{e_1 \downarrow w_1 \ldots e_n \downarrow w_n \quad \mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}} \quad e^{\mathtt{f}}\sigma \downarrow w}{\mathtt{f}(e_1, \cdots, e_n) \downarrow w} \ \text{où } x_i\sigma = w_i \neq \mathbf{Err} \text{ et } w \neq \mathbf{Err}$$

FIG. I.2: Sémantique par appel par valeur d'un programme $\mathbf{p}$

**Définition I.1.** *Un symbole de fonction* $\mathtt{f}$ *d'arité* $n$ *appartenant à un programme* $\mathbf{p}$ *calcule une fonction partielle* $[\![\mathtt{f}]\!] : \mathcal{V}^n \to \mathcal{V}^*$ *définie par : Pour tout* $v_i \in \mathcal{V}$, $[\![\mathtt{f}]\!](v_1, \cdots, v_n) = w$ *si et seulement si* $\mathtt{f}(v_1, \cdots, v_n) \downarrow w$.

On généralise la notation $[\![-]\!]$ aux termes clos, c'est-à-dire aux expressions sans variable. Étant donné un terme clos $t$, on note $[\![t]\!] = u$ si et seulement si $t \downarrow u$.

**Exemple 1** (Division). *Considérons le programme suivant qui calcule la division :*

$$\begin{aligned}
&\mathtt{minus}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of} \\
&\qquad \mathbf{0}, z \to \mathbf{0} \\
&\qquad \mathbf{S}(z), \mathbf{0} \to \mathbf{S}(z) \\
&\qquad \mathbf{S}(u), \mathbf{S}(v) \to \mathtt{minus}(u,v) \\
&\mathtt{q}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of} \\
&\qquad \mathbf{0}, \mathbf{S}(z) \to \mathbf{0} \\
&\qquad \mathbf{S}(z), \mathbf{S}(u) \to \mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u), \mathbf{S}(u)))
\end{aligned}$$

*En utilisant la notation* $\underline{n}$ *pour représenter* $\underbrace{\mathbf{S}(\ldots\mathbf{S}(\mathbf{0})\ldots)}_{n \text{ fois } \mathbf{S}}$*, nous obtenons :*

$$[\![\mathtt{q}]\!](\underline{n}, \underline{m}) = \underline{\lceil n/m \rceil} \text{ for } m > 0$$

**Définition I.2** (Contexte). *Un* contexte *est une expression* $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ *contenant une unique occurrence de chaque* $\diamond_i$ *où les symboles* $\diamond_i$ *sont de nouvelles variables qui n'apparaissent pas dans* $\Sigma$*. La substitution de chaque* $\diamond_i$ *par une expression* $d_i$ *est notée* $\mathsf{C}[d_1, \cdots, d_r]$*.*

**Exemple 2.** *Dans le programme de l'exemple 1, la substitution de* $\diamond$ *par* $\mathtt{minus}(z,u)$ *dans le contexte* $\mathsf{C}[\diamond] = \mathbf{S}(\mathtt{q}(\diamond, \mathbf{S}(u)))$ *est égale à* $\mathsf{C}[\mathtt{minus}(z,u)] = \mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u), \mathbf{S}(u)))$*.*

**Définition I.3** (Expression activée). *Supposons qu'un programme* $\boldsymbol{p}$ *possède une définition de la forme* $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$*.* Une expression $d$ est activée *par* $\mathtt{f}(p_1, \cdots, p_n)$*, où les* $p_i$ *sont des motifs, s'il existe un contexte à un trou* $\mathsf{C}[\diamond]$ *tel que :*

- *Si* $e^{\mathtt{f}}$ *est une expression compositionnelle (i.e. où l'opérateur* **Case** *n'apparaît pas), alors* $e^{\mathtt{f}} = \mathsf{C}[d]$*, et, on a* $p_1 = x_1 \ldots p_n = x_n$*.*
- *Sinon,* $e^{\mathtt{f}} = \mathbf{Case}\ x_1, \cdots, x_n\ \mathbf{of}\ \overline{q_1} \to e^1 \ldots \overline{q_\ell} \to e^\ell$*, il existe une position* $j$ *telle que* $e^j = \mathsf{C}[d]$*. Dans ce dernier cas,* $p_i$ *correspond au i-ème élément de la séquence* $\overline{q_j}$*, pour* $i \in \{1, \ldots, n\}$*.*

**Exemple 3.** *L'expression* $\mathtt{q}(\mathtt{minus}(z,u), \mathbf{S}(u))$ *est activée par* $\mathtt{q}(\mathbf{S}(z), \mathbf{S}(u))$ *dans l'exemple 1.*

Cette définition est utilisée afin de contrôler le flot de données pendant un calcul. En effet, une expression est activée par $\mathtt{f}(p_1, \cdots, p_n)$ lorsque l'on a un appel de la forme $\mathtt{f}(v_1, \cdots, v_n)$ et que chaque $v_i$ est filtré par le motif $p_i$.

**Définition I.4** (Expression maximale). *Une expression* $d$ *activée par* $\mathtt{f}(p_1, \cdots, p_n)$ *est maximale s'il n'y a aucun contexte* $\mathsf{C}[\diamond] \neq \diamond$*, tel que l'expression* $\mathsf{C}[d]$ *soit activée par* $\mathtt{f}(p_1, \cdots, p_n)$*.*

**Exemple 4.** *Dans l'exemple 1, l'expression* $\mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u),\mathbf{S}(u)))$ *est une expression maximale activée par* $\mathtt{q}(\mathbf{S}(z),\mathbf{S}(u))$.

### I.1.3 Précédence

Étant donné un programme, nous définissons une précédence sur les symboles de fonction. Cette précédence sera utilisée tout au long de cette étude.

**Définition I.5** (Précédence)**.** *Étant donné un programme* $\langle Var, Cns, Fct, \mathcal{R}\rangle$*, une précédence* $\geq_{Fct}$ *sur les symboles de fonction découle de la notion d'expression activée. En effet, posons* $\mathtt{f} \geq_{Fct} \mathtt{g}$ *s'il existe une séquence d'expressions* $\overline{e}$ *et une séquence de motifs* $\overline{p}$ *telles que* $\mathtt{g}(\overline{e})$ *est activée par* $\mathtt{f}(\overline{p})$*. La clôture réflexive et transitive de* $\geq_{Fct}$ *est aussi notée* $\geq_{Fct}$*. À présent, nous définissons* $\mathtt{f} \approx_{Fct} \mathtt{g}$ *si* $\mathtt{f} \geq_{Fct} \mathtt{g}$ *et* $\mathtt{g} \geq_{Fct} \mathtt{f}$ *sont vérifiées. Enfin,* $\mathtt{f} >_{Fct} \mathtt{g}$ *si* $\mathtt{f} \geq_{Fct} \mathtt{g}$ *est vérifiée et* $\mathtt{g} \geq_{Fct} \mathtt{f}$ *n'est pas vérifiée.*

Intuitivement, $\mathtt{f} \geq_{Fct} \mathtt{g}$ signifie que le symbole de fonction $\mathtt{f}$ appelle le symbole de fonction $\mathtt{g}$ au cours d'une exécution du programme et $\mathtt{f} \approx_{Fct} \mathtt{g}$ signifie que $\mathtt{f}$ et $\mathtt{g}$ s'appellent de manière récursive.

### I.1.4 Arbre des appels

Désormais, nous allons définir la notion d'arbre des appels (ou call-tree) d'un programme. Un arbre des appels est un arbre dont les noeuds correspondent aux appels de fonctions effectués durant une exécution du programme.

Un *état* est un n-uplet $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ où $\mathtt{f}$ représente un symbole de fonction d'arité $n$ et $u_1, \ldots, u_n$ sont des valeurs. Étant donnés deux états, $\eta_1 = \langle \mathtt{f}, u_1, \cdots, u_n\rangle$ et $\eta_2 = \langle \mathtt{g}, v_1, \cdots, v_k\rangle$, et une expression $\mathsf{C}[\mathtt{g}(e_1, \cdots, e_k)]$ activée par $\mathtt{f}(p_1, \cdots, p_n)$, il y a une *transition* entre les deux états $\eta_1$ et $\eta_2$, notée $\eta_1 \rightsquigarrow \eta_2$, s'il existe une substitution $\sigma$ telle que :

1. $p_i\sigma = u_i$ pour $i = 1, \ldots, n$
2. et $[\![e_j\sigma]\!] = v_j$ pour $j = 1, \ldots, k$.

L'ensemble des transitions apparaissant durant l'exécution d'un programme constitue un arbre, dont les noeuds sont les états, et que l'on appelle arbre des appels de $\mathtt{f}$ pour les valeurs $u_1, \ldots, u_n$ si $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ est sa racine. La notion d'état est confondue avec celle de stack frame puisqu'elle contient un appel de fonction ainsi que les valeurs qui lui sont passées comme arguments. Un arbre des appels de racine $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ représente tous les stack frames qui seront empilés sur la pile pendant le calcul de $\mathtt{f}(u_1, \ldots, u_n)$. Nous utiliserons parfois la notation $\overset{*}{\rightsquigarrow}$ pour désigner la clôture réflexive et transitive de $\rightsquigarrow$ et la notation $\overset{+}{\rightsquigarrow}$ pour désigner la clôture transitive de $\rightsquigarrow$.

**Exemple 5.** *Un arbre des appels correspondant au programme de l'exemple 1 est fourni dans la figure I.3.*

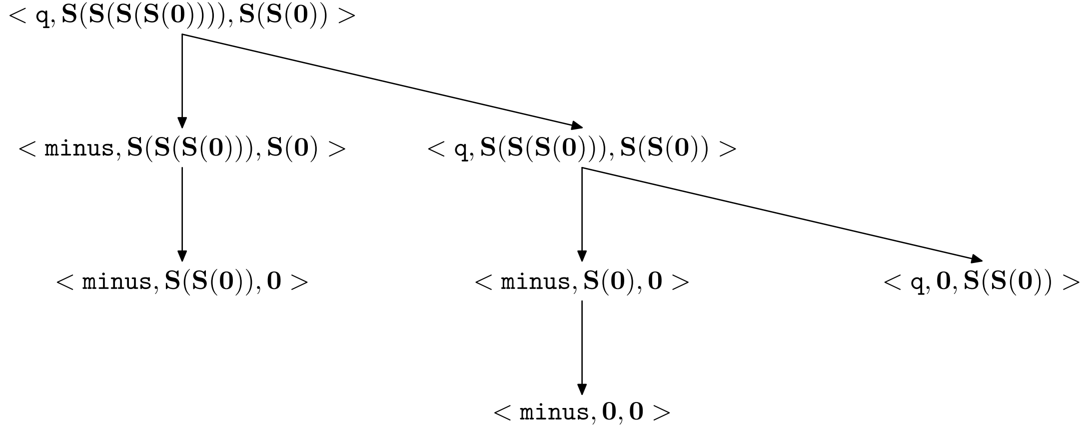


Fig. I.3: Arbre des appels correspondant à l'évaluation de $\mathtt{q}(\mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{0})))), \mathbf{S}(\mathbf{S}(\mathbf{0})))$

### I.1.5 Assignations partielles

**Définition I.6.** *Étant donné un ensemble $\mathbb{K}$, une assignation $I$ est une application partielle de domaine $dom(I)$ inclus dans le vocabulaire $\Sigma$ qui associe une fonction partielle $I(b)$ de $\mathbb{K}^n$ dans $\mathbb{K}$ à chaque symbole $b$ d'arité $n$ appartenant au domaine $dom(I)$. Afin de simplifier le présent discours, nous supposerons désormais qu'une assignation partielle est toujours définie pour les symboles de constructeur (i.e. $Cns \subseteq dom(I)$).*

*Une assignation $I$ est définie sur une expression $e$ si chaque symbole de $Cns \cup Fct$ apparaissant dans l'expression $e$ appartient au domaine $dom(I)$. Si l'assignation $I$ est définie sur une expression $e$ ayant $m$ variables alors l'assignation partielle de $e$ par rapport à $I$, que nous notons $I^*(e)$, est l'extension canonique de l'assignation $I$. Elle dénote une fonction de $\mathbb{K}^m$ dans $\mathbb{K}$ et est définie comme suit :*

1. *Si $x_i$ appartient à Var, alors $I^*(x_i) = X_i$ avec $X_1, \ldots, X_m$ une séquence de nouvelles variables à valeur dans $\mathbb{K}$.*
2. *Si $b$ est un symbole d'arité nulle, alors $I^*(b) = I(b)$.*
3. *Si $b$ est un symbole d'arité $n > 0$ et si $e_1, \cdots, e_n$ sont des expressions, alors $I^*(b(e_1, \cdots, e_n)) = I(b)(I^*(e_1), \ldots, I^*(e_n))$*

*Si $\overline{e}$ est une séquence d'expressions $e_1, \cdots, e_k$ alors $I^*(\overline{e}) = I^*(e_1), \ldots, I^*(e_k)$.*

*Étant donné un contexte $\mathsf{C}[\diamond_1, \cdots, \diamond_l]$, $I^*(\mathsf{C})$ est défini comme la fonction de $\mathbb{K}^l \to \mathbb{K}$ qui satisfait, pour toute séquence d'expressions $e_1, \cdots, e_l$, $I^*(\mathsf{C})(\theta^*(e_1), \ldots, \theta^*(e_l)) = I^*(\mathsf{C}[e_1, \cdots, e_l])$.*

**Définition I.7.** *Étant donné un monoïde* $(\mathbb{K}, +)$*, l'ensemble* **Max-Plus** $\{\mathbb{K}\}$ *est défini comme étant la clôture par composition de l'ensemble des fonctions constantes dans* $\mathbb{K}$*, des projections et des opérations* max*,* $+$*. Une assignation* $I$ *est dite max-plus dans* $\mathbb{K}$ *si pour tout symbole* $b$ *appartenant à dom(*$I$*),* $I(b)$ *est une fonction de* **Max-Plus** $\{\mathbb{K}\}$*.*

**Définition I.8.** *Étant donné un semi-anneau* $(\mathbb{K}, +, \times)$*, l'ensemble* **Max-Poly** $\{\mathbb{K}\}$ *est défini comme étant la clôture par composition de l'ensemble des fonctions constantes dans* $\mathbb{K}$*, des projections et des opérations* max*,* $+$*,* $\times$*. Une assignation* $I$ *est dite max-polynomiale dans* $\mathbb{K}$ *si pour tout symbole* $b$ *appartenant à dom(*$I$*),* $I(b)$ *est une fonction de* **Max-Poly** $\{\mathbb{K}\}$*.*

**Définition I.9** (Assignation polynomiale)**.** *Une assignation partielle* $I$ *est polynomiale si pour tout symbole* $b$ *appartenant à dom(*$I$*),* $I(b)$ *est une fonction de* **Max-poly** $\{\mathbb{R}^+\}$*.*

**Définition I.10** (Assignation additive)**.** *L'assignation* $I$ *d'un symbole de constructeur* $\mathbf{c}$ *d'arité* $n$ *est* additive *si :*

$$I(\mathbf{c})(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}} \text{ où } \alpha_{\mathbf{c}} \geq 1 \qquad \text{lorsque } n > 0$$

$$I(\mathbf{c}) = 0 \qquad \text{sinon.}$$

*Si l'assignation de chaque symbole de constructeur d'arité strictement positive est additive alors l'assignation est additive.*

**Définition I.11.** *La taille d'une expression* $e$*, notée* $|e|$*, est définie par* $|e| = 0$ *si* $e$ *est un symbole d'arité nulle et* $|b(e_1, \ldots, e_n)| = 1 + \sum_i |e_i|$ *si* $e = b(e_1, \ldots, e_n)$ *avec* $n > 0$*.*

**Lemme I.12.** *Étant donnée une assignation additive* $I$*, il existe une constante* $\alpha$ *telle que pour toute valeur* $v$ *de* $\mathcal{V}^*$*, les inégalités suivantes sont satisfaites :*

$$|v| \leq I^*(v) \leq \alpha \times |v|$$

*Démonstration.* Posons $\alpha = \max_{\mathbf{c} \in Cns}(\beta_{\mathbf{c}})$ où la constante $\beta_{\mathbf{c}}$ est choisie comme étant égale à la constante $\alpha_{\mathbf{c}}$ de la définition I.10, si $\mathbf{c}$ est d'arité strictement positive, et comme étant égale à $I^*(\mathbf{c})$ dans le cas contraire. Les inégalités sont obtenues directement par induction sur la taille d'une valeur. □

### I.1.6 Quasi-interprétations

**Définition I.13.** *Étant donné un programme* $\mathbf{p}$*, une quasi-interprétation est une assignation totale* $(\!|-|\!)$ *(i.e. définie pour chaque symbole du programme) qui est monotone et possède la propriété sous-terme. En d'autres termes, pour tout symbole* $b$ *d'arité* $n$*, on a :*

– $\forall i \in \{1, \dots, n\}\,,\ X_i \geq Y_i \Rightarrow (\!|b|\!)(X_1, \cdots, X_n) \geq (\!|b|\!)(Y_1, \cdots, Y_n)$ *(Monotonie)*

– $\forall i \in \{1, \dots, n\}\,, (\!|b|\!)(\dots, X_i, \dots) \geq X_i$ *(Sous-terme)*

*De plus, cette assignation doit vérifier pour toute expression maximale e activée par* $\mathtt{f}(p_1, \cdots, p_n)$ *:*

$$(\!|\mathtt{f}(p_1, \cdots, p_n)|\!)^* \geq (\!|e|\!)^*$$

*Une quasi-interprétation est dite polynomiale ou additive si l'assignation correspondante est polynomiale, respectivement, additive.*

**Exemple 6.** *Le programme suivant calcule un logarithme en base deux :*

$$\begin{aligned}
\mathtt{half}(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{0} &\rightarrow \mathbf{0} \\
\mathbf{S}(\mathbf{0}) &\rightarrow \mathbf{0} \\
\mathbf{S}(\mathbf{S}(y)) &\rightarrow \mathbf{S}(\mathtt{half}(y)) \\
\mathtt{log}(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{0} &\rightarrow \mathbf{0} \\
\mathbf{S}(\mathbf{0}) &\rightarrow \mathbf{0} \\
\mathbf{S}(\mathbf{S}(y)) &\rightarrow \mathbf{S}(\mathtt{log}(\mathbf{S}(\mathtt{half}(y))))
\end{aligned}$$

*Il admet la quasi-interprétation additive et polynomiale suivante :*

$$\begin{aligned}
(\!|\mathbf{0}|\!) &= 0 \\
(\!|\mathbf{S}|\!)(X) &= X + 1 \\
(\!|\mathtt{log}|\!)(X) &= (\!|\mathtt{half}|\!)(X) = X
\end{aligned}$$

*En effet, on vérifie aisément que cette assignation est monotone, sous-terme et additive. Il reste à vérifier les inégalités pour chaque expression activée :*

$$\begin{aligned}
(\!|\mathtt{half}(\mathbf{0})|\!)^* &= 0 \\
&\geq (\!|\mathbf{0}|\!) \\
(\!|\mathtt{half}(\mathbf{S}(\mathbf{0}))|\!)^* &= (\!|\mathbf{S}(\mathbf{0})|\!)^* \\
&= 1 + (\!|\mathbf{0}|\!) \\
&\geq (\!|\mathbf{0}|\!)
\end{aligned}$$

$$\begin{aligned}
(\!|\texttt{half}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= Y+2 \\
&\geq Y+1 \\
&= (\!|\mathbf{S}(\texttt{half}(y))|\!)^* \\
(\!|\texttt{log}(\mathbf{0})|\!)^* &= 0 \\
&\geq (\!|\mathbf{0}|\!) \\
(\!|\texttt{log}(\mathbf{S}(\mathbf{0}))|\!)^* &= 1 \\
&\geq (\!|\mathbf{0}|\!) \\
(\!|\texttt{log}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= Y+2 \\
&\geq (\!|\mathbf{S}(\texttt{log}(\mathbf{S}(\texttt{half}(y))))|\!)^*
\end{aligned}$$

**Exemple 7.** *Le programme suivant correspond à l'algorithme du tri par insertion sur des entiers unaires :*

$$\begin{aligned}
\mathbf{if}(x,y,z) = \mathbf{Case}\ x,y,z\ \mathbf{of}\ \mathbf{True}, v, w &\to v \\
\mathbf{False}, v, w &\to w \\
\mathbf{le}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\ \mathbf{0}, \mathbf{S}(v) &\to \mathbf{True} \\
w, \mathbf{0} &\to \mathbf{False} \\
\texttt{insert}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\ u, \mathbf{nil} &\to \mathbf{c}(u, \mathbf{nil}) \\
u, \mathbf{c}(v,l) &\to \texttt{if}(\mathbf{le}(u,v), \mathbf{c}(u, \mathbf{c}(v,l)), \mathbf{c}(v, \texttt{insert}(u,l))) \\
\texttt{sort}(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{nil} &\to \mathbf{nil} \\
\mathbf{c}(u,l) &\to \texttt{insert}(u, \texttt{sort}(l))
\end{aligned}$$

*Il admet la quasi-interprétation additive et polynomiale suivante :*

$$\begin{aligned}
(\!|\mathbf{0}|\!) = (\!|\mathbf{nil}|\!) = (\!|\mathbf{True}|\!) = (\!|\mathbf{False}|\!) &= 0 \\
(\!|\mathbf{if}|\!)(X,Y,Z) &= \max(X,Y,Z) \\
(\!|\leq|\!)(X,Y) &= \max(X,Y) \\
(\!|\mathbf{S}|\!)(X) &= X+1 \\
(\!|\mathbf{c}|\!)(X,Y) = (\!|\texttt{insert}|\!)(X,Y) &= X+Y+1 \\
(\!|\texttt{sort}|\!)(X) &= X
\end{aligned}$$

*Il suffit de vérifier que les inégalités des quasi-interprétations sont bien satisfaites.*

## I.2 Propriétés des quasi-interprétations

Nous commencerons par donner quelques rappels de complexité sur les Machines de Turing [Tur36]. Puis, nous démontrons une série de lemmes utilisés afin de prouver le lemme fondamental. Ce lemme peut être interprété de la manière suivante « Tout programme admettant une quasi-interprétation additive et polynomiale possède une borne supérieure polynomiale sur la taille des valeurs qu'il calcule en fonction de la taille de ses entrées ». Combiné à des ordres récursifs sur les chemins, ce lemme va permettre de caractériser la classe FPtime des fonctions calculables en temps polynomial ainsi que la classe FPspace des fonctions calculables en espace polynomial.

### I.2.1 Machines de Turing et classes de complexité polynomiales

Nous rappelons brièvement les notions de fonctions calculables en temps polynomial et en espace polynomial en introduisant le modèle de calcul discret qu'est la Machine de Turing. Pour plus d'informations sur les Machines de Turing, nous invitons le lecteur à consulter l'ouvrage suivant [Sip96] ainsi que la référence historique [Tur36]. Une Machine de Turing est constituée d'un alphabet, d'un ou plusieurs rubans infinis, de têtes de lecture et d'écriture ainsi que d'un dispositif de contrôle. Au début d'un calcul, l'entrée est placée sur le premier ruban tandis que tous les autres rubans sont vierges. L'un de ces rubans est appelé ruban de sortie. La tête de lecture transmet le symbole en cours de lecture sur le ruban au dispositif de contrôle qui choisit alors les opérations à effectuer. Ces opérations consistent à écrire un nouveau symbole en lieu et place du symbole qui est en cours de lecture ou encore à déplacer, si nécessaire, la tête de lecture sur la gauche ou sur la droite du ruban. Enfin, le dispositif de contrôle peut choisir d'accéder à un état particulier qui indique la fin du calcul. Dans cette configuration, les symboles stockés sur le ruban de sortie correspondent au calcul de la machine. On dit alors que la machine s'arrête. Cette description se traduit formellement par :

**Définition I.14** (Machine de Turing). *Une Machine de Turing est un sextuplet*

$$(Q, \Sigma, \Gamma, \delta, q_i, q_f)$$

$Q, \Sigma, \Gamma$ *sont des ensembles finis qui représentent respectivement les états du dispositif de contrôle, l'alphabet d'entrée et l'alphabet de travail. On a* $\Sigma \cup \{\mathtt{B}\} \subseteq \Gamma$*, où* $\mathtt{B}$ *est un symbole blanc n'appartenant pas à* $\Sigma$*.*

- $\delta$ *est une fonction de transition de* $Q \times \Gamma$ *dans* $Q \times \Gamma \times \{G, D, 0\}$ *qui a l'état en cours et à un symbole en cours de lecture associe un nouvel état, un symbole à écrire*

*ainsi qu'un déplacement sur le ruban (G code un déplacement à gauche, D code un déplacement à droite et* 0 *indique qu'il n'y a aucun déplcament) .*
- $q_i$ *est l'état initial.*
- $q_f$ *est l'état final.*

**Définition I.15** (Configuration)**.** *Une configuration d'une Machine de Turing à* $k$ *rubans est donnée par le* $k+1$ *uplet* $(q, w_1\#v_1, \ldots w_k\#v_k)$ *où* $q$ *est l'état en cours,* $\#$ *est un symbole spécial n'appartenant pas à l'alphabet de travail et les* $w_i$ *correspondent aux symboles du i-ème ruban situés à gauche de la tête de lecture tandis que les* $v_i$ *correspondent aux symboles du i-ème ruban situés à droite de la tête de lecture en incluant le symbole en cours de lecture. La taille* $|C|$ *d'une configuration* $C = (q, w_1\#v_1, \ldots w_k\#v_k)$ *est définie comme le nombre maximal de symboles contenus sur l'un des rubans, i.e.* $|C| = \max_{i\in\{1,k\}}(|v_i| + |w_i|)$*, où la taille d'un mot* $|w|$ *est définie comme étant égale au nombre de symboles de l'alphabet le composant.*

**Définition I.16** (Turing calculabilité)**.** *Une fonction* $\mathtt{f}$ *de* $\Sigma^*$ *dans* $\Gamma^*$ *est Turing calculable s'il existe une Machine de Turing* $(Q, \Sigma, \Gamma, \delta, q_i, q_f)$ *qui s'arrête sur l'entrée* $s \in \Sigma^*$ *si et seulement si* $\mathtt{f}(s)$ *est définie, et dans ce cas, la valeur* $\mathtt{f}(s)$ *est stockée sur le ruban de sortie,.i.e. il existe t configurations* $C_1, \cdots, C_t$ *telles que* $C_1 = (q_i, \epsilon\#s, \epsilon, \ldots, \epsilon) \to \ldots \to C_t = (q_f, w_1\#v_1, \ldots w_k\#v_k)$*, où* $\epsilon$ *représente le mot vide, où* $\to$ *désigne une application de la fonction de transition* $\delta$ *et où* $w_iv_i = \mathtt{f}(s)$ *si le i-ème ruban est le ruban de sortie.*

**Définition I.17** (FPtime)**.** *Une fonction totale* $\mathtt{f}$ *de* $\Sigma^*$ *dans* $\Gamma^*$ *est calculable en temps polynomial s'il existe un polynôme* $P$ *et une Machine de Turing* $M$ *tels que, pour tout* $s \in \Sigma^*$*,* $M$ *calcule* $\mathtt{f}(s)$ *en utilisant un nombre de configurations t borné par* $P(|s|)$*.* FPtime *est défini comme l'union des ensembles de fonctions calculables en temps polynomial pour tout polynôme* $P$*.*

**Définition I.18** (FPspace)**.** *Une fonction totale* $\mathtt{f}$ *de* $\Sigma^*$ *dans* $\Gamma^*$ *est calculable en espace polynomial s'il existe un polynôme* $P$ *et une Machine de Turing* $M$ *tels que, pour tout* $s \in \Sigma^*$*,* $M$ *calcule* $\mathtt{f}(s)$ *en utilisant des configurations dont la taille est bornée par* $P(|s|)$ *; c'est-à-dire pour chaque configuration* $C$ *on a* $|C| \leq P(|s|)$*.* FPspace *est défini comme l'union des ensembles de fonctions calculables en espace polynomial pour tout polynôme* $P$*.*

### I.2.2 Lemme fondamental

Soit **p** un programme admettant une quasi-interprétation, on commence par montrer que toute valeur $u$ calculée à partir d'un terme clos $t$ possède une quasi-interprétation inférieure à celle de $t$.

**Proposition I.19.** *Soit $\boldsymbol{p}$ un programme admettant une quasi-interprétation $(\!|-|\!)$, alors pour tout terme clos $t$ tel que $t \downarrow u$, on a :*

$$(\!|t|\!)^* \geq (\!|u|\!)^*$$

*Démonstration.* On montre ce résultat par induction sur la longueur de l'évaluation. Supposons que le terme clos $e$ soit de la forme $b(e_1, \cdots, e_n)$. Par hypothèse d'induction (H.I.), si $e_i \downarrow u_i$ alors $(\!|e_i|\!)^* \geq (\!|u_i|\!)^*$. De plus, supposons que $b(u_1, \cdots, u_n) \downarrow u$ alors, par hypothèse d'induction, on a $(\!|b(u_1, \cdots, u_n)|\!)^* \geq (\!|u|\!)^*$. Ainsi, on obtient :

$$\begin{aligned}
(\!|e|\!)^* &= (\!|b(e_1, \cdots, e_n)|\!)^* && \text{Puisque } e = b(e_1, \cdots, e_n) \\
&= (\!|b|\!)((\!|e_1|\!)^*, \ldots, (\!|e_n|\!)^*) && \text{Par définition de } (\!|-|\!)^* \\
&\geq (\!|b|\!)((\!|u_1|\!)^*, \ldots, (\!|u_n|\!)^*) && \text{Par H.I.et monotonie} \\
&= (\!|b(u_1, \cdots, u_n)|\!)^* && \text{Par définition de } (\!|-|\!)^* \\
&\geq (\!|u|\!)^* && \text{Par H.I.}
\end{aligned}$$

□

Puis, on montre, comme dans [BMM01, BMM07, MM00], que la quasi-interprétation borne la taille des valeurs calculées :

**Proposition I.20.** *Soit $\boldsymbol{p}$ un programme admettant une quasi-interprétation additive $(\!|-|\!)$, alors pour tout terme clos $t$ tel que $t \downarrow u$, on a :*

$$(\!|t|\!)^* \geq |u|$$

*Démonstration.* Ce résultat s'obtient directement en combinant le lemme I.12 et la proposition I.19. □

En particulier, on obtient le résultat suivant lorsque l'on considère un symbole de fonction ainsi qu'une quasi-interprétation additive :

**Corollaire I.21.** *Étant donné un programme $\boldsymbol{p}$ admettant une quasi-interprétation additive $(\!|-|\!)$, pour tout symbole de fonction $\mathtt{f}$ de $\boldsymbol{p}$ et pour toutes valeurs $v_1, \cdots, v_n \in \mathcal{V}$, si $[\![\mathtt{f}(v_1, \cdots, v_n)]\!]$ est défini, i.e. $[\![\mathtt{f}(v_1, \cdots, v_n)]\!] \in \mathcal{V}$, alors on a :*

$$(\!|\mathtt{f}|\!)((\!|v_1|\!)^*, \ldots, (\!|v_n|\!)^*) \geq |[\![\mathtt{f}(v_1, \cdots, v_n)]\!]|$$

**Lemme I.22.** *Étant donné un programme $\boldsymbol{p}$ admettant une quasi-interprétation additive et polynomiale $(\!|-|\!)$, alors pour tout symbole de fonction $\mathtt{f}$, il existe un polynôme $P_{\mathtt{f}}$ tel que, pour toutes valeurs $v_1, \cdots, v_n \in \mathcal{V}$, si $[\![\mathtt{f}(v_1, \cdots, v_n)]\!]$ est défini, i.e. $[\![\mathtt{f}(v_1, \cdots, v_n)]\!] \in \mathcal{V}$, alors on a :*

$$P_{\mathtt{f}}(\max(|v_1|, \ldots, |v_n|)) \geq |[\![\mathtt{f}(v_1, \cdots, v_n)]\!]|$$

*Démonstration.* On définit le polynôme $R_{\mathtt{f}}$ par $R_{\mathtt{f}}(X) = (\!|\mathtt{f}|\!)(\alpha \times X, \dots, \alpha \times X)$ avec $\alpha$ la constante du lemme I.12. On obtient alors :

$$\begin{aligned}
R_{\mathtt{f}}(\max(|v_1|, \dots, |v_n|)) &= (\!|\mathtt{f}|\!)(\alpha \times \max_{i=1..n}(|v_i|), \dots, \alpha \times \max_{i=1..n}(|v_i|)) & \\
&\geq (\!|\mathtt{f}|\!)(\alpha \times |v_1|, \dots, \alpha \times |v_n|) & \text{Monotonie de } (\!|\mathtt{f}|\!) \\
&\geq (\!|\mathtt{f}|\!)((\!|v_1|\!)^*, \dots, (\!|v_n|\!)^*) & \text{Lemme I.12} \\
&\geq |[\![\mathtt{f}(v_1, \cdots, v_n)]\!]| & \text{Corollaire I.21}
\end{aligned}$$

$R_{\mathtt{f}}$ est une fonction de **Max-Poly** $\{\mathbb{R}^+\}$. On peut donc aisément trouver un polynôme $P_{\mathtt{f}}$ qui la borne. □

Ce résultat se généralise aux expressions dans le « lemme fondamental ».

**Lemme I.23** (Lemme fondamental)**.** *Soit **p** un programme admettant une quasi-interprétation polynomiale et additive* $(\!|-|\!)$*, il existe un polynôme $P$ tel que pour toute expression $t$ ayant $n$ variables $x_1, \cdots, x_n$ et pour toute substitution $\sigma$ vérifiant $x_i\sigma = v_i$, on ait :*

$$|[\![t\sigma]\!]| \leq P^{|t|}(\max_{i=1..n} |v_i|)$$

*où $P^1(X) = P(X)$ et $P^{k+1}(X) = P(P^k(X))$.*

*Démonstration.* Étant donné un symbole de constructeur $\mathbf{c}$ d'arité $n > 0$, on définit $P_{\mathbf{c}}(X)$ comme étant égal à $n \times X + 1$. Si $n = 0$ alors on pose $P_{\mathbf{c}}(X) = 0$. Étant donné un symbole de fonction $\mathtt{f}$, $P_{\mathtt{f}}$ est le polynôme associé à $\mathtt{f}$ défini dans le lemme I.22. À présent, soit $P = \sum_{b \in Cns \cup Fct}(P_b)$, nous allons montrer ce résultat par induction sur la structure d'une expression $e = b(e_1, \cdots, e_n)$. À cet effet, supposons par hypothèse d'induction (H.I.) que pour tout $j$, $|[\![t_j\sigma]\!]| \leq P^{|t_j|}(\max_{i=1..n} |v_i|)$, on obtient :

$$\begin{aligned}
|[\![t\sigma]\!]| &= |[\![b([\![e_1\sigma]\!], \dots, [\![e_n\sigma]\!])]\!]| & \\
&\leq P_b(\max(|[\![e_1\sigma]\!]|, \dots, |[\![e_n\sigma]\!]|)) & \text{Lemme I.22} \\
&\leq P(\max(|[\![e_1\sigma]\!]|, \dots, |[\![e_n\sigma]\!]|)) & \text{Définition de } P \\
&\leq P(\max(P^{|t_1|}(\max_{i=1..n} |v_i|), \dots, P^{|t_n|}(\max_{i=1..n} |v_i|))) & \text{H.I.} \\
&\leq P^{|t|}(\max_{i=1..n} |v_i|) & \text{Propriété sous-terme de } P
\end{aligned}$$

□

Par la suite, la borne de complexité ainsi fournie dépendra uniquement de la taille des entrées puisque le terme $t$ est fixé. Le lemme I.23 et la propriété sous-terme des quasi-interprétations impliquent que chaque appel intermédiaire effectué pendant un

calcul est évalué sur des valeurs dont la taille est bornée polynomialement par la taille des entrées. De tels résultats auraient pu être obtenus à l'aide d'interprétations polynomiales [Lan79, MN70]. Cependant, contrairement aux interprétations polynomiales, la quasi-interprétation ne fournit plus aucune garantie sur la terminaison des programmes. Ceci nous motive à étudier la combinaison des quasi-interprétations avec des ordres de terminaison.

### I.2.3 Ordres récursifs sur les chemins

Étant donnés un programme **p** et une *précédence* (quasi-ordre) $\geq_{\mathcal{F}}$ sur les symboles de fonction et sa clôture transitive, que nous noterons aussi $\geq_{\mathcal{F}}$ par abus de langage, nous utiliserons la notation $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ si et seulement si $\mathtt{f} \geq_{\mathcal{F}} \mathtt{g}$ et $\mathtt{g} \geq_{\mathcal{F}} \mathtt{f}$ ainsi que la notation $\mathtt{f} >_{\mathcal{F}} \mathtt{g}$ si et seulement si on a $\mathtt{f} \geq_{\mathcal{F}} \mathtt{g}$ mais pas $\mathtt{g} \geq_{\mathcal{F}} \mathtt{f}$. On associe à chaque symbole de fonction $\mathtt{f}$ un statut $st(\mathtt{f})$ à valeur dans $\{p, l\}$ qui vérifie si $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ alors $st(\mathtt{f}) = st(\mathtt{g})$. Le statut indique comment comparer les appels récursifs d'un programme. Lorsque $st(\mathtt{f}) = p$, le statut de $\mathtt{f}$ est appelé statut produit. Dans ce cas de figure, les arguments d'un appel récursif sont comparés avec l'extension produit de $\prec_{rpo}$. Sinon, le statut est lexicographique.

**Remarque 1.** *Par la suite, il est nécessaire de bien distinguer la précédence $\geq_{Fct}$ de la définition I.5, qui désigne une précédence particulière fixée par les définitions d'un programme, de la précédence $\geq_{\mathcal{F}}$ qui désigne une précédence quelconque.*

**Définition I.24.** *L'extension produit $\prec^p$ et l'extension lexicographique $\prec^l$ d'un ordre $\prec$ se définissent de la manière suivante :*

- *Extension produit : $\{m_1, \cdots, m_k\} \prec^p \{n_1, \cdots, n_k\}$ si et seulement si (i) $\forall i \leq k, m_i \preceq n_i$ et (ii) $\exists j \leq k$ tel que $m_j \prec n_j$.*
- *Extension lexicographique : $\{m_1, \cdots, m_k\} \prec^l \{n_1, \cdots, n_l\}$ si et seulement si $\exists j$ tel que $\forall i < j,\ m_i \preceq n_i$ et $m_j \prec n_j$*

**Définition I.25.** *Étant donnés une précédence $\geq_{\mathcal{F}}$ et un statut st, on définit l'ordre récursif sur les chemins $\prec_{rpo}$ à l'aide des règles de la figure I.25. Un programme est ordonné par $\prec_{rpo}$ s'il existe une précédence $\geq_{\mathcal{F}}$ et un statut st tels que pour toute expression r activée par l, on ait $l \succ_{rpo} r$.*

**Théorème I.26** ([Der82, KL80])**.** *Un programme ordonné par $\prec_{rpo}$ termine.*

Les ordres récursifs sur les chemins (RPO) ont été initialement introduits pour permettre de démontrer la terminaison de programmes de manière automatique. En effet, il

$$\frac{\{t_1, \cdots, t_n\} \prec^p_{rpo} \{u_1, \cdots, u_n\}}{\mathbf{c}(t_1, \cdots, t_n) \prec_{rpo} \mathbf{c}(u_1, \cdots, u_n)} \; \mathbf{c} \in Cns$$

$$\frac{u = t_i \text{ or } u \prec_{rpo} t_i}{u \prec_{rpo} b(\ldots, t_i, \ldots)} \; b \in Cns \cup Fct$$

$$\frac{\forall i \; u_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathbf{c}(u_1, \cdots, u_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \; \mathbf{c} \in Cns, \; \mathtt{f} \in Fct$$

$$\frac{\forall i \; u_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{f} >_{\mathcal{F}} \mathtt{g}}{\mathtt{g}(u_1, \cdots, u_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \; \mathtt{g}, \mathtt{f} \in Fct$$

$$\frac{\{u_1, \cdots, u_n\} \prec^{st(\mathtt{f})}_{rpo} \{t_1, \cdots, t_n\} \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad \forall i \; u_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(u_1, \cdots, u_n) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \; \mathtt{g}, \mathtt{f} \in Fct$$

Fig. I.4: Définition de $\prec_{rpo}$

a été démontré dans [KN85] que le problème consistant à déterminer si un programme est ordonné par RPO est NP-complet. Cependant, cette technique est désormais supplantée par d'autres techniques d'analyse de la terminaison des programmes comme les paires de dépendance [AG00] qui sont plus facilement automatisables. Les ordres récursifs sur les chemins démontrent tout de même la terminaison d'un grand nombre d'algorithmes naturels et peuvent être utilisés afin d'étudier la complexité des programmes. L'ordre produit que nous avons introduit ci-dessus est un cas particulier d'un ordre plus général sur les multi-ensembles défini par :

**Définition I.27.** *L'extension multi-ensemble $\prec^m$ d'un ordre $\prec$ est définie par $N = \{s_1, \cdots, s_k\} \prec^m \{r_1, \cdots, r_m\} = M$ s'il existe $i \in \{1, k\}$ tel que $\{r_{i_1}, \ldots, r_{i_k}\} \prec s_i$, avec $r_{i_1}, \ldots, r_{i_k} \subseteq M$, et que $M - \{r_{i_1}, \ldots, r_{i_k}\} \preceq^m N - \{s_i\}$.*

Ces ordres nous permettent d'obtenir les résultats suivants :

**Théorème I.28** ([Hof92])**.** *L'ensemble des fonctions calculées par un programme ordonné par $\prec_{rpo}$ avec statut multi-ensemble est exactement l'ensemble des fonctions primitives récursives.*

**Théorème I.29** ([Wei95])**.** *L'ensemble des fonctions calculées par un programme ordonné par $\prec_{rpo}$ avec statut lexicographique est exactement l'ensemble des fonctions mul-*

*tiples récursives.*

### I.2.4 Une caractérisation de FPtime

Combinés à des quasi-interprétations polynomiales et additives, les ordres récursifs sur les chemins permettent de caractériser l'ensemble des fonctions calculables en temps polynomial.

**Théorème I.30** ([MM00]). *L'ensemble des fonctions calculées par des programmes admettant une quasi-interprétation polynomiale et additive et ordonnés par $\prec_{rpo}$ où chaque symbole a un statut produit est exactement l'ensemble des fonctions calculables en temps polynomial.*

*Démonstration.* La preuve est entièrement écrite dans [BMM07]. Nous en donnons ici les principaux ingrédients.

- Nous commençons par montrer que toute fonction calculée par un programme admettant une quasi-interprétation polynomiale et additive et ordonné par $\prec_{rpo}$ où chaque symbole a un statut produit est inclus dans l'ensemble des fonctions calculables en temps polynomial. L'ordre $\prec_{rpo}$ avec un statut produit, implique que chaque appel récursif de la forme $\mathtt{f}(v_1, \cdots, v_n)$, avec $\mathtt{f}$ un symbole de fonction et des valeurs $v_i$, sera effectué sur des sous-termes des $v_i$. Nous pouvons facilement démontrer la propriété suivante

$$(v_1, \cdots, v_n) \prec^p_{rpo} (u_1, \cdots, u_n) \Rightarrow (|v_1|, \cdots, |v_n|) <^p (|u_1|, \cdots, |u_n|) \tag{I.1}$$

  Il résulte de ceci que l'on a au plus $\max_{i=1}^n(|v_i|)$ appels récursifs consécutifs à l'exécution de $\mathtt{f}(v_1, \cdots, v_n)$. En utilisant des techniques de memoïzation à la Jones [Jon97], qui consistent à stocker dans un cache toutes les valeurs intermédiaires calculées, on définit un interpréteur d'appel par valeur en figure I.5. La mémoïzation correspond à des techniques de programmation dynamique. Si l'appel $\mathtt{f}(v_1, \cdots, v_n)$ retourne la valeur $v$ alors on ajoute le 2-uplet $(\mathtt{f}(v_1, \cdots, v_n), v)$ au cache. Si au cours de l'évaluation du programme, un nouvel appel de la forme $\mathtt{f}(v_1, \cdots, v_n)$ est effectué, alors on regarde dans le cache si la valeur correspondante a déjà été calculée. Dans le cas contraire, on continue l'évaluation en ajoutant la nouvelle valeur obtenue dans le cache. Il reste donc à montrer que la taille du cache est bien bornée polynomialement par la taille des entrées. Ce résultat découle du lemme I.23 combiné à l'inégalité I.1.
- La complétude est démontrée dans le théorème I.35 en utilisant des machines à registres (Register Machine ou RM).

$$\frac{x\sigma = w}{R, \sigma \vdash \langle C, x\rangle \rightarrow \langle C, w\rangle}\ (Variable) \qquad \frac{\mathbf{c} \in Cns \quad R, \sigma \vdash \langle C_{i-1}, t_i\rangle \rightarrow \langle C_i, w_i\rangle}{R, \sigma \vdash \langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \rightarrow \langle C_n, \mathbf{c}(w_1, \cdots, w_n)\rangle}\ (Cons)$$

$$\frac{\mathtt{f} \in Fct \quad R, \sigma \vdash \langle C_{i-1}, t_i\rangle \rightarrow \langle C_i, w_i\rangle \quad (\mathtt{f}(w_1, \cdots, w_n), w) \in C_n}{R, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \rightarrow \langle C_n, w\rangle}\ (LectureCache)$$

$$\frac{R, \sigma \vdash \langle C_{i-1}, t_i\rangle \rightarrow \langle C_i, w_i\rangle \quad \mathtt{f}(\overline{x}) = \mathbf{Case}\ \overline{x}\ \mathbf{of}\ \overline{p} \rightarrow e \quad p_i\sigma' = w_i \quad R, \sigma' \vdash \langle C_n, e\rangle \rightarrow \langle C, w\rangle}{R, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \rightarrow \langle C \cup (\mathtt{f}(w_1, \cdots, w_n), w), w\rangle}\ (Push)$$

FIG. I.5: Évaluation d'un programme à l'aide d'un interpréteur avec cache

□

Nous commençons par donner la définition d'une RM :

**Définition I.31** (RM)**.** *Une RM est un modèle de calcul sur l'algèbre des mots binaires* $\{0,1\}^*$ *qui se compose :*

- *D'un ensemble fini d'états* $S = \{s_1, \cdots, s_k\}$ *incluant un état de départ* `BEGIN`.
- *D'un ensemble fini de registres* $\Pi = \{\pi_1, \cdots, \pi_m\}$ *stockant des valeurs dans* $\{0,1\}^*$. *Le dernier registre* $\pi_m$ *est appelé registre de sortie.*
- *D'une fonction de transition* `com` *associant à chaque état l'une des commandes suivantes* $\mathbf{Succ}(\pi = i(\pi), s')$, *pour* $i = 0, 1$, $\mathbf{Pred}(\pi = p(\pi), s')$, $\mathbf{Branch}(\pi, s', s'')$ *et* **End**.

*Une configuration d'une RM est définie par un couple* $(s, F)$ *ou* $s$ *est l'état en cours d'exécution et* $F$ *est une fonction qui à chaque registre* $\pi_j$ *associe une valeur* $w_j$ *de* $\{0,1\}^*$. *Par la suite, étant donnés un registre* $\pi$ *de* $\Pi$ *et une valeur* $w \in \{0,1\}^*$, *on utilisera la notation* $F\{\pi \leftarrow w\}$ *pour signifier que la valeur de* $\pi$ *est mise à jour à la valeur* $w$ *dans* $F$.

À présent, nous définissons la sémantique d'une telle machine :

**Définition I.32.** *La sémantique d'une RM est donnée par une fonction partielle* `eval` *de* $\mathbb{N} \times S \times (\{0,1\}^*)^m$ *dans* $\{0,1\}^*$ *définie par :*

- $\mathtt{eval}(0, s, F) = \perp$.
- *Si* $\mathtt{com}(s) = \mathbf{Succ}(\pi = i(\pi), s')$, $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', F\{\pi \leftarrow i(F(\pi))\})$.
- *Si* $\mathtt{com}(s) = \mathbf{Pred}(\pi = p(\pi), s')$, $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', F\{\pi \leftarrow p(F(\pi))\})$ *avec* $p(i(w)) = w$ *pour* $i = 0, 1$.

– *Si* $\texttt{com}(s) = \textbf{Branch}(\pi, s', s'')$ *alors* $\texttt{eval}(t+1, s, F) = \texttt{eval}(t, r, F)$ *avec* $r = s'$ *si* $F(\pi) = 0(w)$ *et* $r = s''$ *si* $F(\pi) = 1(w)$.
– *Si* $\texttt{com}(s) = \textbf{End}$ *alors* $\texttt{eval}(t+1, s, F) = F(\pi_m)$.

On définit désormais la notion de calculabilité pour une RM :

**Définition I.33.** *Étant donnée une fonction $T$ de $\mathbb{N}$ dans $\mathbb{N}$, on dit qu'une fonction $\phi$ de $(\{0,1\}^*)^n$ dans $\{0,1\}^*$ est calculée par une RM en temps $T$ s'il existe une RM à $m$ registres avec $m \geq n$ telle que :*

$$\forall (w_1, \cdots, w_n) \in (\{0,1\}^*)^n, \ \texttt{eval}(T(\max_{i=1}^{n}(|w_i|)), \textit{BEGIN}, F) = \phi(w_1, \cdots, w_n)$$

*avec* $\forall i \in \{1, \ldots, n\}, \ F(\pi_i) = w_i$ *et* $\forall i \in \{n+1, \ldots, m\}, \ F(\pi_i) = \epsilon$.

*Une fonction $\phi$ est calculée par une RM en temps polynomial s'il existe un polynôme $P$ et une RM $M$ tels que $M$ calcule $\phi$ en temps $P$.*

D'après [Pap94], on sait qu'une Machine de Turing peut être simulée en temps linéaire par une RM :

**Théorème I.34.** *Une fonction $\phi$ est calculable en temps polynomial si et seulement si elle est calculée par une RM en temps polynomial.*

Nous allons donc démontrer que toute fonction calculée par une RM en temps polynomial peut être simulée par un programme de notre langage terminant par $\prec_{rpo}$ avec un statut produit et admettant une quasi-interprétation polynomiale et additive.

**Théorème I.35** ([MM00])**.** *Supposons que la fonction $\phi$ de $\{0,1\}^*$ dans $\{0,1\}^*$ est calculée par une RM en temps polynomial $T$, alors elle peut être calculée par un programme ordonné par $\prec_{rpo}$ avec statut produit et admettant une quasi-interprétation polynomiale et additive.*

*Démonstration.* On présente le programme calculant $\phi'$ qui utilise un constructeur $\mathbf{c}$ pour encoder les états :

$$\begin{aligned}
\texttt{step}(x) = \textbf{Case}\ x\ \textbf{of}\ \mathbf{c}(s, \pi_1, \cdots, \pi_m) &\rightarrow \mathbf{c}(s', \pi_1, \ldots, \mathbf{i}(\pi_j), \ldots, \pi_m) \\
&\qquad \text{Si } \texttt{com}(s) = \textbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s') \\
\mathbf{c}(s, \pi_1, \ldots, i(\pi_j), \ldots, \pi_m) &\rightarrow \mathbf{c}(s', \pi_1, \ldots, \pi_j, \ldots, \pi_m) \\
&\qquad \text{Si } \texttt{com}(s) = \textbf{Pred}(\pi_j = p(\pi_j), s')
\end{aligned}$$

$$\begin{aligned}
\mathbf{c}(s,\pi_1,\ldots,\mathbf{0}(\pi_j),\ldots,\pi_m) &\to \mathbf{c}(s',\pi_1,\cdots,\pi_m)\\
&\quad \text{Si } \mathtt{com}(s)=\mathbf{Branch}(\pi_j,s',s'')\\
\mathbf{c}(s,\pi_1,\ldots,\mathbf{1}(\pi_j),\ldots,\pi_m) &\to \mathbf{c}(s'',\pi_1,\cdots,\pi_m)\\
&\quad \text{Si } \mathtt{com}(s)=\mathbf{Branch}(\pi_j,s',s'')\\
\mathbf{c}(s,\pi_1,\ldots,\mathbf{0}(\pi_j),\ldots,\pi_m) &\to \pi_m\\
&\quad \text{Si } \mathtt{com}(s)=\mathbf{End}\\
\mathbf{i}(x) &\to \mathbf{i}(x),\ \text{ pour } \mathbf{i}=\mathbf{0},\mathbf{1}
\end{aligned}$$

On achève l'évaluation par la définition suivante :

$$\begin{aligned}
\mathtt{eval}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\ & \epsilon, z \to z\\
& \mathbf{S}(t), z \to \mathtt{step}(\mathtt{eval}(t,z))
\end{aligned}$$

Ce programme termine bien par $\prec_{rpo}$ avec statut produit. De plus, il admet la quasi-interprétation polynomiale et additive $(\!|-|\!)$ définie par $(\!|\epsilon|\!) = 0$, $(\!|\mathbf{S}|\!)(X)=(\!|\mathbf{0}|\!)(X)=(\!|\mathbf{1}|\!)(X) = X+1$, $(\!|\mathbf{c}|\!)(S,X_1,\cdots,X_m) = S+\sum_{i=1}^m X_i + 1$, $(\!|\mathtt{step}|\!)(X) = X+1$ et $(\!|\mathtt{eval}|\!)(X,Y) = X+Y$. Nous laissons le lecteur vérifier que tout polynôme $T$ peut être calculé à l'aide d'un programme sur les entiers unaires (codés à l'aide du symbole de constructeur $\mathbf{S}$) admettant une quasi-interprétation polynomiale et additive. □

### I.2.5 Une caractérisation de FPSPACE

À présent, nous donnons une caractérisation des fonctions calculables en espace polynomial à l'aide de quasi-interprétations polynomiales et additives et de l'ordre de terminaison $\prec_{rpo}$.

**Théorème I.36** ([BMM01]). *L'ensemble des fonctions calculées par des programmes admettant une quasi-interprétation polynomiale et additive et ordonnés par $\prec_{rpo}$ est exactement l'ensemble des fonctions calculables en espace polynomial.*

*Démonstration.* La preuve présentée ci-dessous provient de [BMM07].

– Nous commençons par montrer que toute fonction calculée par un programme admettant une quasi-interprétation polynomiale et additive et ordonné par $\prec_{rpo}$ est inclus dans l'ensemble des fonctions calculables en espace polynomial. L'ordre $\prec_{rpo}$, implique la propriété suivante :

$$(v_1,\cdots,v_n) \prec^l_{rpo} (u_1,\cdots,u_n) \Rightarrow (|v_1|,\cdots,|v_n|) <^l (|u_1|,\cdots,|u_n|)$$

Il résulte de ceci que l'on a au plus $\prod_{i=1}^{n}(|v_i|)$ appels récursifs lors de l'exécution de $\mathtt{f}(v_1, \cdots, v_n)$. Puisque le nombre de symboles de fonction est fixé par la taille du programme, en appliquant le lemme I.23, on obtient que la taille de chaque branche du call-tree est bornée polynomialement par la taille des entrées. En utilisant une stratégie d'évaluation qui consiste en un parcours en profondeur de l'arbre des appels, on obtient que tout programme ordonné par $\prec_{rpo}$ admettant une interprétation polynomiale et additive correspond à une fonction pouvant être calculée par une Machine de Turing en espace polynomial.

- La réciproque est démontrée dans le théorème I.41.

□

Nous commençons par donner la définition d'une PRM (Parallel Register Machine) :

**Définition I.37** (PRM)**.** *Une PRM est un modèle de calcul sur l'algèbre des mots binaires qui se compose d'une RM à laquelle on ajoute les commandes* $\mathbf{Fork}_{\min}(s', s'')$ *et* $\mathbf{Fork}_{\max}(s', s'')$*. Une configuration d'une PRM est définie de manière analogue à la configuration d'une RM.*

À présent, nous définissons la sémantique d'une telle machine :

**Définition I.38.** *La sémantique d'une PRM est donnée par une fonction partielle* $\mathtt{eval}$ *de* $\mathbb{N} \times S \times (\{0,1\}^*)^m$ *dans* $\{0,1\}^*$ *définie dans la définition I.32 et étendue aux commandes* $\mathbf{Fork}_{\min}$ *et* $\mathbf{Fork}_{\max}$ *de la manière suivante :*

- $\mathtt{eval}(t+1, s, F) = \mathtt{min}_{\mathtt{W}}(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$, *si* $com(s) = \mathbf{Fork}_{\min}(s', s'')$
- $\mathtt{eval}(t+1, s, F) = \mathtt{max}_{\mathtt{W}}(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$, *si* $com(s) = \mathbf{Fork}_{\max}(s', s'')$

*où les opérations* $\mathtt{min}_{\mathtt{W}}$ *et* $\mathtt{max}_{\mathtt{W}}$ *sont définies sur les mots binaires par :*

$$\begin{aligned}
\mathtt{min}_{\mathtt{W}}(\epsilon, w) &= \epsilon & \mathtt{max}_{\mathtt{W}}(\epsilon, w) &= w \\
\mathtt{min}_{\mathtt{W}}(w, \epsilon) &= \epsilon & \mathtt{max}_{\mathtt{W}}(w, \epsilon) &= w \\
\mathtt{min}_{\mathtt{W}}(0(w_1), 1(w_2)) &= 0(w_1) & \mathtt{max}_{\mathtt{W}}(0(w_1), 1(w_2)) &= 1(w_2) \\
\mathtt{min}_{\mathtt{W}}(1(w_1), 0(w_2)) &= 0(w_2) & \mathtt{max}_{\mathtt{W}}(1(w_1), 0(w_2)) &= 1(w_1) \\
\mathtt{min}_{\mathtt{W}}(i(w_1), i(w_2)) &= i(\mathtt{min}_{\mathtt{W}}(w_1, w_2)) & \mathtt{max}_{\mathtt{W}}(i(w_1), i(w_2)) &= i(\mathtt{max}_{\mathtt{W}}(w_1, w_2))
\end{aligned}$$

On définit désormais la notion de calculabilité pour une PRM :

**Définition I.39.** *Étant donnée une fonction* $T$ *de* $\mathbb{N}$ *dans* $\mathbb{N}$*, on dit qu'une fonction* $\phi$ *de* $(\{0,1\}^*)^n$ *dans* $\{0,1\}^*$ *est calculée par une PRM en temps* $T$ *s'il existe une PRM à* $m$ *registres, avec* $m \geq n$ *telle que :*

$$\forall (w_1, \cdots, w_n) \in (\{0,1\}^*)^n,\ \mathtt{eval}(T(\max_{i=1}^{n}(|w_i|)), \mathit{BEGIN}, F) = \phi(w_1, \cdots, w_n)$$

*avec* $\forall i \in \{1, \ldots, n\}\,,\ F(\pi_i) = w_i$ *et* $\forall i \in \{n+1, \ldots, m\}\,,\ F(\pi_i) = \epsilon$

*Une fonction $\phi$ est calculée par une PRM en temps polynomial s'il existe un polynôme $P$ et une PRM $M$ tels que $M$ calcule $\phi$ en temps $P$.*

D'après [Pap94], on sait qu'il existe une relation entre le temps de cacul d'une PRM et l'espace de calcul d'une Machine de Turing :

**Théorème I.40.** *Une fonction $\phi$ est calculable en espace polynomial si et seulement si elle est calculée par une PRM en temps polynomial.*

Nous allons donc démontrer que toute fonction calculée par une PRM en temps polynomial peut être simulée par un programme de notre langage terminant par $\prec_{rpo}$ et admettant une quasi-interprétation polynomiale et additive.

**Théorème I.41** ([BMM01])**.** *Supposons que la fonction $\phi$ de $\{0,1\}^*$ dans $\{0,1\}^*$ est calculée par une PRM en temps polynomial $T$, alors elle peut être calculée par un programme ordonné par $\prec_{rpo}$ et admettant une quasi-interprétation polynomiale et additive.*

*Démonstration.* On présente le programme calculant $\phi'$ :

$$
\begin{aligned}
&\texttt{eval}(x, y, x_1, \cdots, x_m) = \textbf{Case } x, y, x_1, \cdots, x_m \textbf{ of}\\
&\quad \mathbf{S}(t), s, \pi_1, \cdots, \pi_m \to \texttt{eval}(t, s', \pi_1, \ldots, \mathbf{i}(\pi_j), \ldots, \pi_m)\\
&\qquad\qquad \text{Si } \texttt{com}(s) = \textbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s')\\
&\quad \mathbf{S}(t), s, \pi_1, \ldots, \mathbf{i}(\pi_j), \ldots, \pi_m \to \texttt{eval}(t, s', \pi_1, \ldots, \pi_j, \ldots, \pi_m)\\
&\qquad\qquad \text{Si } \texttt{com}(s) = \textbf{Pred}(\pi_j = p(\pi_j), s')\\
&\quad \mathbf{S}(t), s, \pi_1, \ldots, \mathbf{0}(\pi_j), \ldots, \pi_m \to \texttt{eval}(t, s', \pi_1, \cdots, \pi_m)\\
&\qquad\qquad \text{Si } \texttt{com}(s) = \textbf{Branch}(\pi_j, s', s'')\\
&\quad \mathbf{S}(t), s, \pi_1, \ldots, \mathbf{1}(\pi_j), \ldots, \pi_m \to \texttt{eval}(t, s'', \pi_1, \cdots, \pi_m)\\
&\qquad\qquad \text{Si } \texttt{com}(s) = \textbf{Branch}(\pi_j, s', s'')\\
&\quad \mathbf{S}(t), s, \pi_1, \ldots, \pi_m \to \mathit{max}_{\mathtt{W}}(\texttt{eval}(t, s', \pi_1, \cdots, \pi_m), \texttt{eval}(t, s'', \pi_1, \cdots, \pi_m))\\
&\qquad\qquad \text{Si } \texttt{com}(s) = \textbf{Fork}_{\max}(s', s'')
\end{aligned}
$$

$$\mathbf{S}(t), s, \pi_1, \ldots, \pi_m \rightarrow \mathit{min}_{\mathtt{W}}(\mathtt{eval}(t, s', \pi_1, \cdots, \pi_m), \mathtt{eval}(t, s'', \pi_1, \cdots, \pi_m))$$
$$\text{Si } \mathtt{com}(s) = \mathbf{Fork}_{\min}(s', s'')$$
$$\mathbf{S}(t), s, \pi_1, \ldots, 0(\pi_j), \ldots, \pi_m \rightarrow \pi_m$$
$$\text{Si } \mathtt{com}(s) = \mathbf{End}$$
$$\epsilon, s, \pi_1, \ldots, \pi_m \rightarrow \perp$$
$$\mathtt{f}(t, x) = \mathtt{eval}(t, \mathtt{BEGIN}, x, \epsilon, \ldots, \epsilon)$$

D'abord, constatons que $\mathit{max}_{\mathtt{W}}$ et $\mathit{min}_{\mathtt{W}}$ peuvent être calculés par un programme ordonné par $\prec_{rpo}$ et admettant une quasi-interprétation additive et polynomiale. En effet, il suffit de substituer le symbole $\rightarrow$ au symbole $=$ dans les règles de la définition I.38, puis, de choisir $(\!|\epsilon|\!) = 0$, $(\!|\mathbf{0}|\!)(X) = (\!|\mathbf{1}|\!)(X) = X+1$ et $(\!|\mathit{min}_{\mathtt{W}}|\!)(X,Y) = (\!|\mathit{max}_{\mathtt{W}}|\!)(X,Y) = \max(X,Y)$ tout en vérifiant que ce programme est bien ordonné par $\prec_{rpo}$. Le programme décrit ci-dessus termine bien par $\prec_{rpo}$. De plus, il admet la quasi-interprétation polynomiale et additive $(\!|-|\!)$ étendue par $(\!|\mathbf{S}|\!)(X) = X+1$, $(\!|\mathtt{eval}|\!)(Z, S, X_1, \cdots, X_m) = Z + S + \sum_{i=1}^{m} X_i + 1$, $(\!|\mathtt{f}|\!)(Z, X) = Z + T + m$. Nous laissons au lecteur le soin de vérifier que tout polynôme $T$ peut être calculé à l'aide d'un programme sur les entiers unaires (codés à l'aide du symbole de constructeur $\mathbf{S}$) admettant une quasi-interprétation polynomiale et additive. □

## I.3 Synthèse de quasi-interprétations

La synthèse de quasi-interprétations est un problème incontournable dans une perspective d'application pratique des quasi-interprétations. Ce problème a été introduit par Amadio dans [Ama05]. Il s'agit de trouver une quasi-interprétation pour un programme donné. Un algorithme synthétisant des quasi-interprétations permettrait d'automatiser l'analyse statique des ressources d'un programme. On conjecture que ce problème est indécidable en général et qu'il le demeure si l'on se restreint à des polynômes de degré borné et à coefficients sur les rationnels ou sur les entiers. Cependant, il peut être intéressant de définir de plus petites classes d'assignations pour lesquelles il sera plus aisé de déterminer si un programme admet une quasi-interprétation.

Utilisant les résultats de [BMMP05], nous nous intéressons à la synthèse de quasi-interprétations sur l'ensemble des réels positifs $\mathbb{R}^+$. Plusieurs motivations expliquent un tel choix. Tout d'abord, on peut montrer qu'il existe des programmes n'admettant que des quasi-interprétations possédant au moins un coefficient irrationnel, bien qu'ils soient extrêmement rares en pratique. Ensuite, on démontre que le problème de la synthèse est

décidable en temps exponentiel sur la classe **Max-Poly** $\{\mathbb{R}^+\}$ des assignations de degrés bornés. Ce résultat est une conséquence du théorème de Tarski [Tar51]. En général, la procédure d'élimination des quantificateurs a une complexité doublement exponentielle en le nombre d'alternances entre les blocs de quantificateurs existentiels et universels d'une formule du premier ordre. La complexité de notre problème s'avère être exponentielle car il ne possède qu'une unique alternance. Enfin, la classe des quasi-interprétations **Max-Poly** $\{\mathbb{R}^+\}$ permet de capturer un grand nombre d'algorithmes naturels en pratique.

La complexité de la procédure de synthèse sur **Max-Poly** $\{\mathbb{R}^+\}$ nous incite aussi à considérer de plus petites classes de fonctions. C'est dans cette optique qu'Amadio [Ama05] a considéré des assignations dans **Max-Plus** $\{\mathbb{N}\}$. Amadio a établi que le problème de la synthèse de quasi-interprétation est NPtime-difficile pour les assignations **Max-Plus** $\{\mathbb{N}\}$ de degrés bornés - les différentes notions de degré considérées sont explicitées dans cette section - et NPtime-complet pour les assignations **Max-Plus**-multilinéaires, c'est-à-dire les assignations **Max-Plus** $\{0, 1\}$ de degré borné. On démontre que le problème de la synthèse demeure NPtime-difficile sur **Max-Plus** $\{\mathbb{R}^+\}$ si l'on considère des assignations de degrés bornés. On aurait pu espérer un meilleur résultat en faisant une analogie avec la programmation linéaire qui est Ptime-complète sur $\mathbb{R}^+$ et NPtime-complète sur $\mathbb{N}$. Ce résultat s'inspire de la démonstration de NP-difficulté proposée par Amadio à ceci près que les coefficients de la quasi-interprétation sont à valeur sur $\mathbb{R}^+$ au lieu de $\mathbb{N}$, engendrant ainsi des problèmes de codage. En effet, des propriétés du type « si $x+y \leq 1$ alors soit $x = 1$ et $y = 0$ soit l'inverse », sont valables sur $\mathbb{N}$ mais ne le sont plus sur $\mathbb{R}^+$. Il est donc nécessaire de coder les programmes d'une manière différente. Ceci étant, nous démontrerons que le problème de la vérification, qui consiste à vérifier qu'une assignation donnée est une quasi-interprétation, s'effectue en temps polynomial.

### I.3.1 Définition

Le problème de la synthèse se définit comme suit :

**Définition I.42** (Problème de la synthèse). *Étant donné un programme $\boldsymbol{p}$, existe-t-il une assignation $(\!|-|\!)$ qui est une quasi-interprétation pour $\boldsymbol{p}$ ?*

Nous conjecturons que ce problème est indécidable en général. Il est donc nécessaire de restreindre notre étude à différentes classes de fonctions. Dans cette section, nous nous intéresserons aux classes **Max-Plus** et **Max-Poly** définies dans la section I.1.5. Commençons par démontrer qu'il existe des quasi-interprétations irrationnelles bien qu'elles

soient relativement rares en pratique.

**Définition I.43.** *Une quasi-interprétation de* **Max-Poly** $\{\mathbb{R}\}$ *est irrationnelle si au moins l'un de ses coefficients est irrationnel, i.e. appartient à* $\mathbb{R} - \mathbb{Q}$.

**Proposition I.44.** *Il existe des programmes admettant des quasi-interprétations additives qui sont toutes irrationnelles sur* **Max-Poly** $\{\mathbb{R}\}$.

*Démonstration.* Nous allons construire définition par définition un programme de manière à forcer sa quasi-interprétation $(\!|-|\!) \in$ **Max-Poly** $\{\mathbb{R}\}$ à posséder un coefficient irrationnel. La définition

$$\mathtt{id}(y) = \textbf{Case } y \textbf{ of } \mathbf{a}(x) \to \mathbf{a}(\mathtt{id}(\mathtt{id}(x)))$$

force le symbole de fonction id à posséder une quasi-interprétation $(\!|\mathtt{id}|\!)(X) = X$. On considère un nouveau symbole de fonction g d'arité deux ainsi qu'un symbole de constructeur **0** d'arité nulle vérifiant $(\!|\mathbf{0}|\!) = 0$ sans perte de généralité. On ajoute la définition suivante :

$$\mathtt{id}(y) = \textbf{Case } y \textbf{ of } \mathbf{0} \to \mathtt{g}(\mathbf{0}, \mathbf{0})$$

On obtient l'inégalité ci-dessous :

$$0 \geq (\!|\mathtt{g}|\!)(0, 0)$$

On considère un nouveau symbole de constructeur **b** vérifiant $(\!|\mathbf{b}|\!)(X) = X + k$ avec une constante $k \geq 1$ :

$$\mathtt{id}(y) = \textbf{Case } y \textbf{ of } \mathbf{b}(x) \to \mathtt{g}(\mathbf{b}(\mathbf{0}), x)$$

La quasi-interprétation doit vérifier :

$$X + k \geq (\!|\mathtt{g}|\!)(k, X)$$

Il en découle que $(\!|\mathtt{g}|\!)(X, Y)$ est de degré au plus 1 en $Y$. Sinon, pour $Y$ choisi suffisamment grand, l'inégalité ci-dessus n'est plus vérifiée. Par conséquent, il existe un ensemble $I$ et des fonctions $R_i$ et $S_i$ de **Max-Poly** telles que $(\!|\mathtt{g}|\!)(X, Y) = \max_{i \in I}(R_i(X) \times Y + S_i(X))$. En outre, on a $(\!|\mathtt{g}|\!)(0, 0) = \max_{i \in I}(S_i(0)) = 0$. À présent, nous ajoutons les définitions suivantes :

$$\begin{aligned}
\mathtt{id}(y) &= \textbf{Case } y \textbf{ of } \mathbf{b}'(\mathbf{0}) \to \mathtt{g}(\mathbf{0}, \mathbf{S}(\mathbf{0})) \\
&\qquad\qquad \mathbf{S}(\mathbf{S}(\mathbf{0})) \to \mathtt{g}(\mathbf{0}, \mathtt{g}(\mathbf{0}, \mathbf{S}(\mathbf{0}))) \\
\mathtt{g}(x, y) &= \textbf{Case } x, y \textbf{ of } \mathbf{0}, \mathbf{S}(\mathbf{0}) \to \mathbf{b}'(\mathbf{0}) \\
&\qquad\qquad \mathbf{0}, \mathbf{b}'(\mathbf{0}) \to \mathbf{S}(\mathbf{S}(\mathbf{0}))
\end{aligned}$$

Supposons que $(\!|\mathbf{S}|\!)(X) = \alpha + X$ et que $(\!|\mathbf{b'}|\!)(X) = k' + X$ avec $\alpha, k' \geq 1$. Toutes ces définitions nous donnent :

$$k' \geq \max_{i \in I}(R_i(0)) \times \alpha$$
$$2 \times \alpha \geq \max_{i \in I}(R_i(0))^2 \times \alpha$$
$$\max_{i \in I}(R_i(0)) \times \alpha \geq k'$$
$$\max_{i \in I}(R_i(0)) \times k' \geq 2 \times \alpha$$

La deuxième inégalité nous garantit que $2 \geq \max_{i \in I}(R_i(0))^2$ puisque $\alpha \geq 1$. On déduit de la première et de la troisième inégalités que $k' = \max_{i \in I}(R_i(0)) \times \alpha$. En substituant $k'$ dans la dernière inégalité, on obtient $\max_{i \in I}(R_i(0))^2 \times \alpha \geq 2 \times \alpha$ et, par conséquent, $\max_{i \in I}(R_i(0))^2 \geq 2$. Finalement, $\max_{i \in I}(R_i(0)) = \sqrt{2}$ et le programme n'admet que des quasi-interprétations irrationnelles puisqu'il admet, en particulier, la quasi-interprétation suivante :

$$(\!|\mathbf{0}|\!) = 0$$
$$(\!|\mathbf{b}|\!)(X) = X + 1$$
$$(\!|\mathbf{b'}|\!)(X) = X + 2$$
$$(\!|\mathbf{S}|\!)(X) = X + \sqrt{2}$$
$$(\!|\mathtt{id}|\!)(X) = X$$
$$(\!|\mathsf{g}|\!)(X, Y) = ((1 - \sqrt{2}) \times X + \sqrt{2}) \times Y + X$$

□

### I.3.2 Décidabilité du problème de la synthèse sur Max-Poly $\{\mathbb{R}^+\}$

Dans cette section, nous allons montrer que la recherche de quasi-interprétations sur les réels est décidable. Ce résultat est une conséquence du théorème de Tarski [Tar51]. Commençons par normaliser la représentation des fonctions utilisées dans **Max-Poly** afin de simplifier notre étude.

**Proposition I.45** (Normalisation)**.** *Toute fonction $Q$ de* **Max-Poly** *peut s'écrire sous la forme :*

$$Q(X_1, \ldots, X_n) = \max(P_1(X_1, \ldots, X_n), \ldots, P_k(X_1, \ldots, X_n))$$

*où les $P_i$ sont des polynômes.*

Nous définissons désormais deux notions de degré. La première, appelée $\times$-degré, correspond à la puissance d'un polynôme tandis que la seconde, appelée max-degré, correspond à l'arité de la fonction max.

**Définition I.46** (Degrés)**.** *Soit $Q$ une fonction de* **Max-Poly** *d'arité $n$. On dit que le* max-*degré de $Q$ est $k$ si $Q(X_1, \ldots, X_n) = \max(P_1(X_1, \ldots, X_n), \ldots, P_k(X_1, \ldots, X_n))$. Le $\times$-degré de $Q$ se définit comme étant le degré maximal des polynômes $P_1, \ldots, P_k$ où le degré d'un polynôme $X_1^{i_1} X_2^{i_2} ... X_n^{i_n}$ est égal à $\sum_{j=1}^{n} i_j$.*

**Définition I.47.** *On dira d'une assignation* **Max-Poly** *qu'elle appartient à la classe* **(k,d)-Max-Poly** *si son $\times$-degré et son* max-*degré sont bornés respectivement par les constantes $d$ et $k$.*

**Théorème I.48** ([BMMP05])**.** *Étant données deux constantes $k$ et $d$, le problème de la synthèse pour des assignations de* **(k,d)-Max-Poly** *est décidable en temps exponentiel en la taille du programme.*

*Démonstration.* Considérons un programme $\langle Var, Cns, Fct, \mathcal{R} \rangle$. Nous allons démontrer que le problème de la synthèse dans **(k,d)-Max-Poly** se réduit à la satisfaisabilité d'une formule du premier ordre.

**Définition I.49.** *Étant donné un symbole* `f` *d'arité $n$, son assignation de* max-*degré $k$ et de $\times$-degré $d$ est de la forme :*

$$(\!|\mathtt{f}|\!)(X_1, \ldots, X_n) = \max(P[\mathtt{f}, 1](X_1, \ldots, X_n), \ldots, P[\mathtt{f}, k](X_1, \ldots, X_n))$$

*où $P[\mathtt{f}, i]$ sont des polynômes de $\times$-degré au plus $d$. En d'autres termes,*

$$P[\mathtt{f}, i](X_1, \ldots, X_n) = \sum a[\mathtt{f}, i, j_1, \ldots, j_n] X_1^{j_1} \times \cdots \times X_n^{j_n}$$

*avec $1 \leq i \leq k$ et $\sum_{j=1}^{n}(j_\ell) \leq d$ et où les $a[\mathtt{f}, i, j_1, \ldots, j_n]$ sont de nouvelles variables représentant les coefficients multiplicatifs des polynômes.*

**Définition I.50** (Codage de la propriété sous-terme)**.** *Soit* `f` *un symbole d'arité $n$, la propriété sous-terme peut être exprimée par la formule suivante :*

$$S[\mathtt{f}] \;=\; \bigwedge_{j=1}^{n} S[\mathtt{f}, j]$$

*avec $S[\mathtt{f}, j] = \forall X_1, \ldots, X_n \geq 0 : \bigvee_{i=1}^{k} P[\mathtt{f}, i](X_1, \ldots, X_n) \geq X_j$*

**Définition I.51** (Codage de la propriété de monotonie)**.** *Soit* $\mathtt{f}$ *un symbole d'arité* $n$*, la propriété de monotonie peut être exprimée par :*

$$\begin{array}{c} M[\mathtt{f}] = \forall X_1, \ldots, X_n : \forall\, 0 \leq Y_1 \leq X_1, \ldots, 0 \ \leq Y_n \leq X_n : \\ \bigwedge_{j=1..k} \bigvee_{i=1..n} P[\mathtt{f}, i](X_1, \ldots, X_n) \geq P[\mathtt{f}, j](Y_1, \ldots, Y_n) \end{array}$$

**Définition I.52** (Codage de la propriété d'additivité)**.** *Soit* $\mathbf{c}$ *un symbole de constructeur. On exprime l'additivité de l'assignation correspondante par la formule suivante :*

$$\begin{array}{rl} C[\mathbf{c}] = & a[\mathbf{c}, 1, 0, \ldots, 0] \geq 1 \quad \wedge \\ & \bigwedge_{\Sigma j_\ell = 1} a[\mathbf{c}, 1, j_1, \ldots, j_n] = 1 \quad \wedge \\ & \bigwedge_{\Sigma j_\ell > 1} a[\mathbf{c}, 1, j_1, \ldots, j_n] = 0 \quad \wedge \\ & \bigwedge_{i=2..k, j_1, \ldots, j_n} a[\mathbf{c}, i, j_1, \ldots, j_n] = 0 \end{array}$$

**Proposition I.53.** *Étant données une assignation* $(\!|-|\!)$ *de* max*-degré* $k$ *et de* $\times$*-degré* $d$ *et une expression* $t$ *de taille* $m$*, le* max*-degré de* $(\!|t|\!)^*$ *est au plus* $k^m$ *et son* $\times$*-degré est borné par* $d^m$*.*

*Démonstration.* Par induction sur la taille du terme $t$.

Les propositions I.45 et I.53 montre que les polynômes $P[\mathtt{f}, i]$ peuvent être étendus aux expressions. On écrit alors :

$$(\!|t|\!)^*(X_1, \ldots, X_n) = \max(P[t, 1](X_1, \ldots, X_n), \ldots, P[t, k'](X_1, \ldots, X_n))$$

avec $k' \leq k^{|t|}$ et avec le $\times$-degré de $t$ borné par $d^{|t|}$, si l'assignation en considération est de max-degré $k$ et de $\times$-degré $d$.

**Définition I.54** (Codage des quasi-interprétations)**.** *Considérons une assignation* $(\!|-|\!)$ *de* **Max-Poly** *d'un programme* $\boldsymbol{p}$ *de* max*-degré* $k$*. Pour chaque expression maximale* $r$ *activée par* $l$*, on définit :*

$$R[l \to r] = \forall X_1, \ldots X_p \geq 0 : \bigwedge_{j=1..l} \bigvee_{i=1..n} P[l, i](X_1, \ldots, X_p) \geq P[r, j](X_1, \ldots, X_p)$$

*avec* $n \leq k^{|l|}$ *et* $l \leq k^{|r|}$ *(cf. proposition I.53).*

**Proposition I.55.** *Étant donné un programme* $\boldsymbol{p}$ *admettant une quasi-interprétation* $(\!|-|\!)$*, la formule du premier ordre* $R[l \to r]$ *est valide si et seulement si* $(\!|l|\!)^* \geq (\!|r|\!)^*$*.*

En utilisant les définitions I.50, I.51, I.52 et I.54, on introduit la formule du premier ordre[1] suivante :

$$\begin{aligned} F[\langle Var, Cns, Fct, \mathcal{R}\rangle] = \; & \exists_{p \in Cns \cup Fct, 1 \leq i \leq k, 0 \leq j_1, \ldots, j_n \leq d} \, a[p, i, j_1, \ldots, j_n] : \\ & \bigwedge_{\mathbf{c} \in Cns} C[\mathbf{c}] \quad \wedge \\ & \bigwedge_{\mathbf{g} \in Fct} (S[\mathbf{g}] \wedge M[\mathbf{g}]) \quad \wedge \\ & \bigwedge_{l \text{ expression maximale activée par } r} R[l \to r] \end{aligned}$$

En prenant garde à renommer toutes les variables dans les différentes inégalités, on peut skolemizer une telle formule. On obtient alors une nouvelle formule avec une alternance d'un bloc de quantificateurs existentiels, représentant les coefficients multiplicatifs des polynômes, et d'un bloc de quantificateurs universels représentant les variables. Nous utilisons alors un théorème d'élimination des quantificateurs sur les réels [HRS90] reposant sur les résultats de Tarski et prouvant notre résultat de décidabilité. Notons que ce théorème est une amélioration d'un résultat de Collins [Col75] qui avait démontré qu'une telle procédure a une complexité doublement exponentielle en le nombre de variables.

**Théorème I.56** ([HRS90])**.** *Soit un $\boldsymbol{K}$ un anneau intègre inclus dans un corps réel clos $\boldsymbol{R}$. Soit $\phi$ une formule de taille $L$ du langage des corps ordonnés à paramètres dans $\boldsymbol{K}$ supposée sous forme prénexe avec $m$ blocs de quantificateurs où apparaissent $s$ polynômes en $n$ variables à coefficient dans $\boldsymbol{K}$ dont la somme des degrés totaux ($\times$-degré) est inférieure ou égale à $D$. Il existe un algorithme de complexité séquentielle $O(L)D^{n^{O(m)}}$ qui calcule une formule sans quantificateurs équivalente à $\phi$.*

Puisque l'on considère les coefficients multiplicatifs $a[f, i, j_1, \ldots, j_n]$ comme étant des variables, les nouveaux coefficients multiplicatifs sont tous égaux à 1. On choisit alors l'anneau intègre $\mathbf{K}$ comme étant égal à $\mathbb{R}$. Supposons que $\phi$ soit la forme prénexe de $F[\langle Var, Cns, Fct, \mathcal{R}\rangle]$, on sait par hypothèse que le $\times$-degré des polynômes est borné par une constante $d$ et que le max-degré est borné par une constante $k$. Par conséquent, $D \leq d \times L$. Le nombre $m$ d'alternances entre les blocs de quantificateurs dans $\phi$ étant égal à 1, il nous reste à trouver des bornes sur $L$ et sur $n$ qui représentent respectivement la taille de la formule et le nombre de variables. Remarquons que trouver une borne sur $L$ revient à trouver une borne sur le nombre d'inégalités dans $\phi$. Soient $i$ l'arité maximale d'un symbole de fonction, $\Delta$ le cardinal de l'ensemble $Cns \cup Fct$ et $\alpha$ la taille maximale d'une définition.

- La définition I.50 introduit au plus $\Delta \times i \times k$ inégalités et ajoute au plus $\Delta \times i$ variables universelles.

[1] $n$ étant l'arité maximale d'un symbole.

- La définition I.51 introduit au plus $\Delta \times k^2$ inégalités et ajoute au plus $2i \times \Delta$ nouvelles variables universelles.
- La définition I.52 introduit au plus $\Delta \times k \times i^{d+1}$ inégalités et n'ajoute aucune nouvelle variable universelle. En effet, on a $\#\{(j_1, ..., j_n) | \sum_{l=1}^{n} j_l \leq d\} \leq n^{d+1}$ où $\#$ est une fonction calculant le cardinal d'un ensemble.
- La définition I.54 introduit au plus $\Delta \times k^\alpha$ inégalités et ajoute au plus $\Delta \times i$ nouvelles variables universelles.
- La dernière formule n'introduit aucune inégalité et ajoute au plus $\Delta \times k \times i^{d+1}$ variables existentielles.

Finalement, si $P$ est la taille du programme, on a $L = O(\Delta \times (i \times k + k^2 + k \times i^{d+1} + k^\alpha)) = O(P \times k^P)$, pour $k$ choisi suffisamment grand, et $n = O(\Delta \times (4i + k \times i^{d+1})) = O(P^{d+2})$. Ainsi, puisque $D \leq d \times L$, en appliquant le théorème I.56, on peut éliminer les quantificateurs au cours d'une procédure dont la complexité est en $O(P \times k^P) \times (O(P \times k^P))^{O(P^{d+2})^{O(1)}}$. Puisque l'élimination des quantificateurs consiste en l'élimination des variables, cette procédure rend un résultat au problème de la synthèse, s'il en existe un. □

Le théorème ci-dessus nous fournit une procédure dont la complexité dépend de la taille du programme, du $\times$-degré et du max-degré afin de trouver une quasi-interprétation pour un programme donné. Par conséquent, si l'on s'intéresse à des classes de polynômes dont les $\times$-degré et max-degré sont fixés, on obtient une procédure de synthèse de quasi-interprétations exponentielle en la taille du programme.

### I.3.3 Synthèse de quasi-interprétations Max-Plus

Dans cette section, nous allons considérer une sous-classe de **Max-Poly**, la classe **Max-Plus** étudiée par Amadio dans [Ama05]. La complexité de la procédure de synthèse décrite dans la section précédente motive amplement une telle restriction.

**Définition I.57.** *On définit le +-degré d'un polynôme comme étant son plus grand coefficient multiplicatif. Si un polynôme de $\times$-degré $l$ s'écrit sous la forme*

$$\sum_{\substack{i_1 + \ldots + i_n = k \\ k \leq l}} \alpha_{(i_1, \cdots, i_n)} \times X_1^{i_1} \times \ldots \times X_n^{i_n}$$

*alors son +-degré est égal à* $\max_{\substack{i_1 + \ldots + i_n = k \\ k \leq l}} (\alpha_{(i_1, \cdots, i_n)})$.

**Définition I.58.** *On dira d'une assignation* **Max-Plus** *qu'elle appartient à la classe* **(k,d)-Max-Plus** *si le +-degré de chacun de ses polynômes est borné la constante $d$ et*

*son* max-*degré est borné par la constant* $k$*.*

**Théorème I.59** ([Ama05])**.** *Le problème de la synthèse de quasi-interprétation est* NPtime*-difficile sur des assignations* **(k,d)-Max-Plus** $\{\mathbb{N}\}$*.*

**Théorème I.60** ([Ama05])**.** *Le problème de la synthèse de quasi-interprétation est* NPtime*-complet sur des assignations* **Max-Plus** $\{0, 1\}$*.*

À présent, nous démontrons que le problème de la synthèse de quasi-interprétation est NPtime-difficile pour des assignations **Max-Plus** sur l'ensemble des réels positifs $\mathbb{R}^+$ de $+$-degré et de max-degré bornés.

**Théorème I.61** ([Péc05])**.** *Le problème de la synthèse de quasi-interprétation est un problème difficile de* NPtime *sur des assignations* **(k,d)-Max-Plus** $\{\mathbb{R}^+\}$*.*

*Démonstration.* La preuve de NPtime-difficulté est démontrée par le théorème I.65. Elle s'inspire largement de celle d'Amadio qui réduit un problème de satisfaction d'une formule 3-CNF ou 3-Forme Normale Conjonctive qui correspond à des disjonctions de conjonctions de 3 littéraux, en un problème de la synthèse pour un programme. Cependant, les deux preuves diffèrent pour des raisons techniques : La preuve d'Amadio utilise des propriétés de l'ensemble des entiers naturels qui ne sont plus valables sur les réels. Par exemple, la propriété suivante est utilisée dans la preuve d'Amadio : $\sum_{i=1}^{n} \alpha_i = 1 \Rightarrow (\exists j$ tel que $\alpha_j = 1$ et $\forall k \neq j\ \alpha_k = 0)$. Notre preuve ajoute de nouvelles définitions afin de compenser la perte de telles propriétés sur les réels. □

De plus, nous démontrerons dans le Théorème I.66, qu'étant donnée une assignation **(k,d)-Max-Plus** $\{\mathbb{R}^+\}$, on peut vérifier en temps polynomial qu'elle est une quasi-interprétation et nous en déduisons que :

**Théorème I.62.** *Le problème de la synthèse de quasi-interprétation est un problème complet de* NPtime *sur des assignations* **(k,d)-Max-Plus** $\{\mathbb{N}\}$*.*

*Démonstration.* La NPtime-difficulté a été démontrée dans le théorème I.59. On montre dans le théorème I.66, qu'étant donnée une assignation candidate, on peut vérifier qu'elle est une quasi-interprétation en temps polynomial (si les variables sont à valeur sur $\mathbb{R}$). Il reste à constater que la taille de toute solution est bornée polynomialement par la taille de l'entrée, puisque ses degrés sont bornés par $k$ et $d$. □

Dans ce qui suit, toute assignation de **Max-Plus** $\{\mathbb{R}^+\}$ sera écrite sous la forme $\max_{i\in I}(\sum_{j=1}^{n} \alpha_{i,j} X_j + a_i)$, à l'aide de la proposition I.45, la taille de l'ensemble $I$ étant bornée par le max-degré et les constantes $\alpha_{i,j}$, $a_i$ de $\mathbb{R}^+$ étant bornées par le $+$-degré.

### NPtime-difficulté

**Proposition I.63.** *Étant donnée une assignation* $\max_{i\in I}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j + a_i)$ *d'un symbole d'arité n, on a :*

$$\textit{Pour tout } j \leq n, \textit{ il existe } i \in I \textit{ tel que } \alpha_{i,j} \geq 1$$

*Démonstration.* Cette propriété est une conséquence directe de la propriétés sous-terme. □

**Proposition I.64.** *Étant donné un programme* **p** *possédant un symbole de fonction* `f` *et admettant une quasi-interprétation* $(\!|-|\!)$, *on peut ajouter des définitions au programme de sorte à ce que l'une des deux conditions suivantes soit vérifiée :*

1. $(\!|\mathtt{f}|\!)(X_1, \ldots, X_n) = \max(X_1, \ldots, X_n)$
2. $(\!|\mathtt{f}|\!)(X_1, \ldots, X_n) = \max_{i\in I}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j)$, *quels que soient les* $\alpha_{i,j}$.

*Démonstration.* On commence par démontrer la première inégalité. Supposons que `f` soit un symbole de fonction possédant une quasi-interprétation de la forme :

$$(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) = \max_{i\in I'}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j + a_i)$$

Considérons la définition suivante :

$$e_1 \equiv \mathtt{f}(x_1, \ldots, x_n) = \mathtt{f}(\mathtt{f}(x_1, \ldots, x_n), \ldots, \mathtt{f}(x_1, \ldots, x_n)) \equiv e_1^{'}$$

Une quasi-interprétation de cette définition doit vérifier $(\!|e_1|\!)^* \geq (\!|e_1^{'}|\!)^*$. Cette condition implique que $(\!|e_1|\!)^* \geq \max_{i\in I'}((\sum_{j=1}^{n} \alpha_{i,j}) \times (\!|e_1|\!)^* + a_i)$. Par conséquent, pour tout $i \in I'$, $\sum_{j=1}^{n} \alpha_{i,j} \leq 1$. Grâce à la proposition I.63, on a pour tout $l$, une constante $k$ pour laquelle $\alpha_{k,l} \geq 1$. Combiné à l'inégalité précédente, ceci implique que $\alpha_{k,i} = 0$ pour tout $i \neq l$.

La quasi-interprétation de `f` s'écrit donc :

$$(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) = \max(X_1 + a_{i_1}, \ldots, X_n + a_{i_n}, \max_{i\in I}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j + a_i))$$

avec $\sum_{j=1}^{n} \alpha_{i,j} \leq 1$ et $I = I' - \{i_1, \cdots, i_n\}$. Pour $X$ choisi suffisamment grand, on a $(\!|\mathtt{f}|\!)(X, 0, \ldots, 0) = X + a_{i_1}$, avec $a_{i_1} \geq 0$. En particulier, cette définition implique que :

$$\begin{aligned}
(\!|\mathtt{f}|\!)(X, 0, \ldots, 0) &= X + a_{i_1} \\
&\geq (\!|\mathtt{f}|\!)((\!|\mathtt{f}|\!)(X, 0, \ldots, 0), \ldots, (\!|\mathtt{f}|\!)(X, 0, \ldots, 0)) \\
&\geq (\!|\mathtt{f}|\!)(X + a_{i_1}, \ldots, X + a_{i_1}) \\
&\geq X + 2 \times a_{i_1}
\end{aligned}$$

et ainsi $a_{i_1} = 0$. On peut obtenir les mêmes résultats pour toute constante $a_{i_k}$. De sorte, que la quasi-interprétation peut s'écrire :

$$(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) = \max(X_1, \ldots, X_n, \max_{i \in I}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j + a_i))$$

avec $\sum_{j=1}^{n} \alpha_{i,j} \leq 1$. Nous ajoutons la définition suivante au programme :

$$\mathtt{f}(\overline{y}) = \textbf{Case } \overline{y} \textbf{ of } \mathbf{d}(x_1, \mathbf{0}), \ldots, \mathbf{d}(x_n, \mathbf{0}) \to \mathtt{f}(\mathbf{d}(x_1, \mathtt{f}(\mathbf{0}, \ldots \mathbf{0})), \ldots \mathbf{d}(x_n, \mathtt{f}(\mathbf{0}, \ldots, \mathbf{0})))$$

Notons $e_2 = \mathtt{f}(\mathbf{d}(x_1, \mathbf{0}), \ldots \mathbf{d}(x_n, \mathbf{0}))$ et $e_2' = \mathtt{f}(\mathbf{d}(x_1, \mathtt{f}(\mathbf{0}, \ldots \mathbf{0})), \ldots \mathbf{d}(x_n, \mathtt{f}(\mathbf{0}, \ldots, \mathbf{0})))$. $(\!|-|\!)$ doit vérifier :

$$\begin{aligned} (\!|e_2|\!)^* &= \max_{i \in I}((\sum_{j=1}^{n} \alpha_{i,j}) \times (X + a^d) + a_i) \\ &\geq \max_{i \in I}((\sum_{j=1}^{n} \alpha_{i,j}) \times (X + a^d + \max_{k \in I}(a_k)) + a_i) \\ &= (\!|e_2'|\!)^* \end{aligned}$$

avec $(\!|\mathbf{d}|\!)(X, Y) = X + Y + a^d$ et pour les valeurs particulières $X_1 = \ldots = X_n = X$. Soit $l$ l'indice pour lequel $\max_{i \in I}(\sum_{j=1}^{n} \alpha_{i,j})$ est atteint. Pour $X$ choisi suffisamment grand et puisque $\sum_{j=1}^{n} \alpha_{l,j} = 1$ et $a_l = 0$, on a $X + a^d \geq X + a^d + \max_{k \in I}(a_k)$. Ce qui est équivalent à $a_k = 0, \ \forall k \in I$. Finalement, on obtient :

$(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) = \max(\max(X_1, \ldots, X_n), \max_{i \in I}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j))$ avec $\sum_{j=1}^{n} \alpha_{i,j} \leq 1$.

Puisque $\forall X_1, \ldots, \forall X_n \in \mathbb{R}^+$, $\max(X_1, \ldots, X_n) \geq \max_{i \in I}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j)$ on obtient :

$$(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) = \max(X_1, \ldots, X_n)$$

Soit $\mathtt{g}$ un symbole de fonction dont la quasi-interprétation vérifie $(\!|\mathtt{g}|\!)(X_1, \cdots, X_n) = \max_{i \in I}(\sum_{j=1}^{n} \alpha_{i,j} \times X_j + a_i)$. Pour prouver la condition 2 de cette proposition, on ajoute encore une autre règle :

$$\mathtt{h}(x) = \textbf{Case } x \textbf{ of } \mathbf{c}(x_1, \ldots, x_n) \to \mathbf{c}(\mathrm{g}(\mathbf{0}, \ldots, \mathbf{0}), \mathbf{0}, \ldots, \mathbf{0})$$

avec $\mathtt{h}$ un symbole de fonction tel que $(\!|\mathtt{h}|\!)(X) = X$ (Une telle fonction existe d'après les définitions précédentes). L'assignation correspondante doit vérifier $a^c + \sum_{j=1}^{n} X_j \geq a^c + \max_{k \in I}(a_k)$ , ce qui est équivalent à $a_k = 0, \ \forall k \in I$. Ainsi, $(\!|\mathtt{g}|\!)(X_1, \cdots, X_n) = \max_{i \in I}(\sum_{j=1}^{n} \alpha_{i,j} X_j)$ et la condition 2 est démontrée. □

**Remarque 2.** *On peut forcer l'égalité des coefficients additifs des symboles de constructeur en ajoutant les définitions suivantes :*

$$\hbar(x) = \textbf{Case } x \textbf{ of } \mathbf{c}(x, \mathbf{0}, \ldots, \mathbf{0}) \to \mathbf{d}(x, \mathbf{0}, \ldots, \mathbf{0})$$
$$\hbar(x) = \textbf{Case } x \textbf{ of } \mathbf{d}(x, \mathbf{0}, \ldots, \mathbf{0}) \to \mathbf{c}(x, \mathbf{0}, \ldots, \mathbf{0})$$

**Théorème I.65.** *Le problème de la synthèse de quasi-interprétation est* NPTIME*-difficile pour des assignations* **(k,d)-Max-Plus** $\{\mathbb{R}^+\}$.

*Démonstration.* D'après la proposition I.64, on peut forcer un symbole de fonction `f` d'arité deux à avoir une quasi-interprétation de la forme suivante :

$$(\!|\mathtt{f}|\!)(X_1, X_2) = \max_{i \in I}(\alpha_{i,1} \times X_1 + \alpha_{i,2} \times X_2)$$

Dans toute cette preuve, nous considèrerons que les symboles de constructeur ont tous le même coefficient additif $k$ dans leurs assignations respectives et on définit $\alpha_j$ comme étant égal à $\max_{i \in I}(\alpha_{i,j})$, pour $j \in \{1, 2\}$, et $\alpha$ comme étant égal à $\max_{i \in I}(\alpha_{i,1} + \alpha_{i,2})$. Ces trois constantes vérifient $\alpha_1 + \alpha_2 \geq \alpha$. Maintenant, ajoutons la définition suivante :

$$\mathtt{f}(x, y) = \textbf{Case } x, y \textbf{ of } \mathbf{d}(x_1, \mathit{0}), \mathbf{d}(x_2, \mathit{0}) \to \mathbf{d}(\mathtt{f}(x_1, \mathit{0}), \mathtt{f}(\mathit{0}, x_1))$$

Pour des valeurs vérifiant $X_1 = X_2$, on obtient que $\alpha \times (X_1 + k) \geq k + (\alpha_1 + \alpha_2) \times X_1$. Et, par conséquent, pour $X_1$ choisi suffisamment grand, $\alpha = \alpha_1 + \alpha_2$. Il résulte de ceci que :

$$(\!|\mathtt{f}|\!)(X_1, X_2) = \alpha_1 \times X_1 + \alpha_2 \times X_2$$

puisqu'il existe $j \in I$ tel que $\alpha_{j,1} = \alpha_1$ et $\alpha_{j,2} = \alpha_2$. Ce qui implique que $\forall X_1, X_2 \in \mathbb{R}^+$, $\forall i \in I$ $\alpha_{j,1} \times X_1 + \alpha_{j,2} \times X_2 \geq \alpha_{i,1} \times X_1 + \alpha_{i,2} \times X_2$. Nous allons démontrer que l'on peut ajouter des définitions au symbole `f` afin de contraindre $\alpha_1$ et $\alpha_2$ à vérifier la condition ci-dessous :

$$(\alpha_1 = 1 \wedge \alpha_2 = 2) \vee (\alpha_1 = 2 \wedge \alpha_2 = 1)$$

En vertu de la condition sous-terme, $\alpha_1, \alpha_2 \geq 1$. On ajoute alors les définitions suivantes :

$$\mathtt{f}(x, y) = \textbf{Case } x, y \textbf{ of } \mathbf{c}(x_1), \mathbf{c}(x_2) \to \mathbf{c}(\mathbf{c}(\mathbf{c}(\mathit{0})))$$
$$\mathtt{h}(x) = \textbf{Case } x \textbf{ of } \mathbf{c}(\mathbf{c}(\mathbf{c}(x))) \to \mathtt{f}(\mathbf{c}(\mathit{0}), \mathbf{c}(\mathit{0}))$$

On en déduit que $\alpha_1 + \alpha_2 = 3$. En ajoutant :

$$\mathtt{h}(\mathbf{c}(\mathbf{c}(x))) \to \mathtt{f}(\mathtt{f}(\mathit{0}, \mathbf{c}(\mathit{0})), \mathit{0})$$

on vérifie que $2\times k+x \geq \alpha_1\times\alpha_2\times k$ avec $\mathtt{h}$ un symbole de fonction vérifiant $(\!|\mathtt{h}|\!)(X) = X$. En d'autres termes, $2 \geq \alpha_1 \times \alpha_2$. Puisque, $\alpha_1 = 3 - \alpha_2$, on veut vérifier cette inégalité $\alpha_1^2 - 3 \times \alpha_1 + 2 \geq 0$ avec $2 \geq \alpha_1 \geq 1$. Les seules solutions de l'inégalité précédente sont $(\alpha_1 = 1 \wedge \alpha_2 = 2) \vee (\alpha_1 = 2 \wedge \alpha_2 = 1)$. Il nous reste alors à encoder la satisfaisabilité d'une formule d'un problème 3-SAT sous forme 3-CNF en un problème de synthèse. Dans ce but, nous associons à chaque littéral $x_i$ apparaissant dans une formule 3-CNF $\phi$ une fonction $\mathtt{f}_i$ telle que $(\!|\mathtt{f}_i|\!)(X_1, X_2) = \alpha_1^i \times X_1 + \alpha_2^i \times X_2$. Le tableau de la figure I.6 résume les différentes valeurs prises par la quasi-interprétation $(\!|\mathtt{f}_i|\!)$ en fonction de ses coefficients et de ses arguments d'entrée :

| Coefficients de $(\!\mid\mathtt{f}_i\mid\!)$ : $(\alpha_1^i, \alpha_2^i)$ | Arguments : $(x_1, x_2)$ | Quasi-interprétation calculée : |
|---|---|---|
| (1,2) | $(\mathbf{c}(\boldsymbol{0}), \boldsymbol{0})$ | $k$ |
| (1,2) | $(\boldsymbol{0}, \mathbf{c}(\boldsymbol{0}))$ | $2 \times k$ |
| (2,1) | $(\mathbf{c}(\boldsymbol{0}), \boldsymbol{0})$ | $2 \times k$ |
| (2,1) | $(\boldsymbol{0}, \mathbf{c}(\boldsymbol{0}))$ | $k$ |

Fig. I.6: Tableau récapitulatif des valeurs de la quasi-interprétation $(\!|\mathtt{f}_i|\!)$

On va supposer que la constante $k$ (respectivement $2 \times k$) permet de coder la valeur de vérité **True** (respectivement **False**). Si un littéral correspond à la valeur de vérité **True** (respectivement **False**) alors on va coder cette information en faisant en sorte que le symbole de fonction correspondant $\mathtt{f}_i$ ait une quasi-interprétation égale à $X_1 + 2 \times X_2$ (respectivement $2 \times X_1 + X_2$). Soit une disjonction $D$ extraite d'une formule $\phi$, il y a deux cas de figure à prendre en considération :

– Si le premier littéral de $D$ est $x_i$, on associe les arguments $(\mathbf{c}(\boldsymbol{0}), \boldsymbol{0})$ au symbole de fonction $\mathtt{f}_i$. Dans ce cas, on a $(\!|\mathtt{f}_i(\mathbf{c}(\boldsymbol{0}), \boldsymbol{0})|\!)^* = \alpha_1 \times k$. Il en résulte que $(\!|\mathtt{f}_i|\!)$ correspondra à la valeur de vérité **True** si et seulement si $\alpha_1 = 1$. C'est à dire, si et seulement si $(\!|\mathtt{f}_i|\!)(X_1, X_2) = X_1 + 2 \times X_2$.
– Si le premier littéral de $D$ est $\neg x_i$, on associe les arguments $(\boldsymbol{0}, \mathbf{c}(\boldsymbol{0}))$ au symbole de fonction $\mathtt{f}_i$. Dans ce cas, on a $(\!|\mathtt{f}_i(\mathbf{c}(\boldsymbol{0}), \boldsymbol{0})|\!)^* = \alpha_2 \times k$. Il en résulte que $(\!|\mathtt{f}_i|\!)$ correspondra à la valeur de vérité **True** si et seulement si $\alpha_2 = 1$. C'est à dire, si et seulement si $(\!|\mathtt{f}_i|\!)(X_1, X_2) = 2 \times X_1 + X_2$.

Utilisons la notation $arg(x, D)$ pour représenter les arguments du symbole de fonction encodant $x$ dans la disjonction $D$ :

$$arg(x, D) = \begin{cases} (\mathbf{c}(\boldsymbol{0}), \boldsymbol{0}) \text{ si } x \text{ apparaît dans } D \\ (\boldsymbol{0}, \mathbf{c}(\boldsymbol{0})) \text{ si } \neg x \text{ apparaît dans } D \end{cases}$$

$(\!|\mathtt{f}(arg(x, D))|\!)^*$ est égal à $k$ si $(\!|\mathtt{f}|\!)$ correspond à la valeur de vérité **True** et si $x$ apparaît dans $D$ ou si $(\!|\mathtt{f}|\!)$ correspond à la valeur de vérité **False** et si $\neg x$ apparaît dans $D$. $(\!|\mathtt{f}(arg(x, D))|\!)^*$ est égal à $2 \times k$ si $(\!|\mathtt{f}|\!)$ correspond à la valeur de vérité **True** et si $\neg x$ apparaît dans $D$ ou si $(\!|\mathtt{f}|\!)$ correspond à la valeur de vérité **False** et si $x$ apparaît dans $D$. Il ne nous reste plus qu'à encoder les disjonctions. Dans cette optique, nous introduisons un symbole de fonction $\mathtt{s}_n$ d'arité $n$ dont la quasi-interprétation vérifie :

$$(\!|\mathtt{s}_n|\!)(X_1, \cdots, X_n) = \alpha_1 \times X_1 + \cdots + \alpha_n \times X_n$$

Une telle quasi-interprétation est obtenue en ajoutant les définitions suivantes au programme :

$$\mathtt{h}(x) = \textbf{Case } x \textbf{ of } \mathbf{c}(x) \to \mathbf{c}(\mathtt{s}_n(\boldsymbol{0}, \ldots, \boldsymbol{0}, x, \boldsymbol{0}, \ldots \boldsymbol{0}))$$

avec $x$ apparaissant à la i-ème position dans le membre droit de la définition pour $i = 1..n$. Finalement, on associe la définition suivante à toutes disjonctions $D$ de la formule $\phi$ contenant les littéraux $x_i$, $x_j$ et $x_k$ :

$$\mathtt{h}(x) = \textbf{Case } x \textbf{ of } \mathbf{c}(\mathbf{c}(\mathbf{c}(\mathbf{c}(\mathbf{c}(x))))) \to \mathtt{s}_3(\mathtt{f}_i(arg(x_i, D), \mathtt{f}_j(arg(x_j, D), \mathtt{f}_k(arg(x_k, D))$$

Sa quasi-interprétation vérifie :

$$5 \times k + X \geq (\!|\mathtt{f}_i(arg(x_i, D))|\!)^* + (\!|\mathtt{f}_j(arg(x_j, D))|\!)^* + (\!|\mathtt{f}_k(arg(x_k, D))|\!)^*$$

Cette inégalité force au moins l'une des quasi-interprétations de symbole de fonction à avoir la valeur $k$ (ce qui correspond à la valeur de vérité **True**). Nous codons de la sorte chacune des disjonctions de la formule $\phi$. Puisque nous avons encodé tout problème 3-CNF en un problème de synthèse de quasi-interprétation sur **Max-Plus** $\{\mathbb{R}^+\}$ et que la satisfaction d'une formule 3-CNF est un problème NPtime-difficile, nous obtenons le résultat escompté. □

### Vérification et NPtime-complétude

Puisque le problème est NPtime-difficile, nous nous intéresserons au problème de la vérification qui consiste à contrôler qu'une assignation est bien une quasi-interprétation. Nous allons démontrer que ce problème appartient à Ptime. De plus, on peut montrer qu'il le demeure si l'on considère des assignations de max-degré $d$ et de +-degré $k$ bornés polynomialement par la taille du programme. Cette condition n'est pas trop restrictive puisque la majorité des programmes admettant une quasi-interprétation dans **Max-Plus** $\{\mathbb{R}^+\}$ la vérifie.

**Théorème I.66** (Vérification)**.** *Supposons que l'on dispose d'un programme $p$ et d'une assignation $(\!|-|\!)$ dans* **Max-Plus** $\{\mathbb{R}^+\}$*, on peut vérifier en temps polynomial en le* max-*degré $d$ et en le $+$-degré si $(\!|-|\!)$ est une quasi-interprétation pour $p$.*

*Démonstration.* Étant donnés un programme $p$ et une assignation $(\!|-|\!)$. Soit $A$ la taille maximale d'une expression apparaissant dans le membre droit d'une définition. $A$ est bornée par la taille du programme. Pour toute définition de la forme $\mathtt{f}(x_1, \cdots, x_n) =$ **Case** $x_1, \cdots, x_n$ **of** $p_1, \ldots, p_n \to e$, on peut calculer $(\!|\mathtt{f}(p_1, \ldots, p_n)|\!)^*$ et $(\!|e|\!)^*$ polynomialement relativement à $A$ et à $d$. Par conséquent, il reste à vérifier que l'inégalité de la quasi-interprétation est vérifiée. Le nombre total de telles inégalités est borné par le nombre de définitions, et, est donc polynomialement borné par la taille du programme. Dans la suite de cette preuve, on note :

$$(\!|\mathtt{f}|\!)(X_1, \ldots, X_n) = \max_{i \in I}(\textstyle\sum_{j=1}^{n} \alpha_{i,j} \times X_j + a_i)$$
$$(\!|e|\!)^* = \max_{k \in L}(\textstyle\sum_{j=1}^{m} \beta_{k,j} \times X_j + b_k)$$

$\{X_1, \ldots, X_m\}$ étant l'ensemble des variables à valeur dans $\mathbb{R}^+$ et correspondant à l'ensemble $\{x_1, \ldots, x_m\}$ des variables apparaissant dans $\mathtt{f}(p_1, \ldots, p_n)$. Par définition du max-degré et d'après la proposition I.53, on a $\sharp I \leq d$ and $\sharp L \leq d^{|e|} \leq d^A$ Le problème de la vérification consiste à prouver que l'inégalité suivante est vérifiée :

$$\max_{i \in I}(\textstyle\sum_{j=1}^{n} \alpha_{i,j} \times (\!|p_j|\!)^* + a_i) \geq \max_{k \in L}(\textstyle\sum_{j=1}^{m} \beta_{k,j} \times X_j + b_k)$$

Si $r$ est le nombre de définitions dans le programme, on élimine l'opérateur max dans le membre droit de l'inégalité. Puisque $\sharp L \leq d^A$, on obtient au plus $r \times d^A$ inégalités de la forme :

$$\max_{i \in I}(\textstyle\sum_{j=1}^{n} \alpha_{i,j} \times (\!|p_j|\!)^* + a_i) \geq \textstyle\sum_{j=1}^{m} \beta_{k,j} \times X_j + b_k$$

Il nous reste à prouver que cette inégalité se vérifie en temps polynomial. Elle est satisfaite dans $\mathbb{R}^+$ si, a contrario, il n'existe pas de valeurs de $X_1, \ldots, X_m$ dans $\mathbb{R}^+$ telles que :

$$\max_{i \in I}(\textstyle\sum_{j=1}^{n} \alpha_{i,j} \times (\!|p_j|\!)^* + a_i) < \textstyle\sum_{j=1}^{m} \beta_{k,j} \times X_j + b_k$$

On obtient donc au plus $r \times d^A$ systèmes d'inégalités :

$$\left\{\textstyle\sum_{j=1}^{n} \alpha_{i,j} \times (\!|p_j|\!)^* + a_i < \textstyle\sum_{j=1}^{m} \beta_{k,j} \times X_j + b_k, i \in I\right\}$$

Puisque $\sharp I \leq d$, la taille de tout système est bornée par $d$. On peut alors résoudre chaque système en temps polynomial en $d$ à l'aide de la programmation linéaire et trouver une solution, s'il en existe une. Dans ce cas, l'assignation n'est pas une quasi-interprétation. En itérant au plus $r \times d^A$ cette résolution de systèmes, on démontre que le problème

de vérification se résoud en temps polynomial pour des assignations de +-degré borné polynomialement en la taille du programme et en le max-degré. □

## I.4 Modularité des quasi-interprétations

Étant donné un opérateur d'union $\cup$ sur les programmes, on dit qu'une propriété $P$ sur les programmes est modulaire relativement à $\cup$ si la proposition suivante est vérifiée :

$$\forall \mathbf{p}_1, \mathbf{p}_2,\ P(\mathbf{p}_1 \cup \mathbf{p}_2) \Leftrightarrow P(\mathbf{p}_1) \wedge P(\mathbf{p}_2)$$

où étant donné un programme $\mathbf{p}$ et une propriété $P$ sur les programmes, $P(\mathbf{p})$ signifie que le programme $\mathbf{p}$ vérifie la propriété $P$.

L'enjeu de la modularité pour les Systèmes de réécriture a été introduit par Toyama dans [Toy87a]. Aujourd'hui, cette approche est classique lorsqu'il s'agit de résoudre des problèmes comme la confluence [Toy87b], la terminaison [KO92, Gra92] ou la complétude [Klo92] en divisant le problème en différentes sous-parties. Le lecteur intéressé devrait consulter les références suivantes Middeldorp [Mid90], Klop [Klo92] et, plus récemment, Ohlebusch [Ohl02] afin d'avoir une vision d'ensemble du problème. Dans ces différentes sources, la modularité est étudiée relativement à la manière dont les programmes sont divisés. Trois principaux types de décomposition d'un programme sont considérés. Le premier est *l'union disjointe* lorsque le programme est divisé en deux sous-programmes qui n'ont aucun symbole en partage. Le deuxième type est *l'union à constructeurs partagés* dans laquelle les fonctions définies dans chaque sous-programme peuvent partager des symboles de constructeur. Enfin, le dernier cas considéré est *l'union hiérarchique*, où les symboles de constructeur de l'un des sous-programmes sont des symboles de fonction de l'autre sous-programme. Les différents types d'unions sont présentés à la figure I.7. Nous présentons dans cette section les résultats obtenus dans [BMP07] permettant d'améliorer l'intensionnalité et la synthèse des quasi-interprétations à l'aide de la modularité.

#### Vers plus d'intensionnalité

On démontre que les quasi-interprétations sont modulaires dans le cas d'une *union disjointe*. En revanche, elles ne le sont pas dans les deux autres cas. Cependant, il est toujours possible de les utiliser afin de prédire une borne supérieure sur les ressources utilisées dans le cas d'une *union à constructeurs partagés*. En outre, après avoir posé quelques restrictions syntactiques, on montre que l'on peut aussi dériver une borne supérieure dans le cas d'une *union hiérarchique*. Une conséquence directe de ces résultats est

que l'on peut analyser les ressources d'un plus grand nombre d'algorithmes. On obtient ainsi des caractérisations des ensembles des fonctions calculables en temps polynomial et en espace polynomial qui possèdent une plus grande complétude intensionnelle.

#### La modularité permet d'améliorer la synthèse de quasi-interprétations

Dans la section précédente, nous avons démontré que le problème de la synthèse de quasi-interprétation, pour un programme ayant $n$ variables, possède une complexité en temps de l'ordre de $O(2^{n^\alpha})$, avec $\alpha$ une constante, pour peu que l'on considère des quasi-interprétations polynomiales de degrés plus petits qu'une constante arbitrairement fixée. D'un côté on a une procédure très générale, d'un autre côté cette procédure a un coût très élevé. Le problème de la modularité des quasi-interprétations est donc un enjeu majeur dès lors que l'on considère des stratégies de type « diviser pour régner » afin de rechercher des quasi-interprétations. Si l'on parvient à diviser un programme ayant $n$ variables en $k$ sous-programmes ayant $n_j$ variables pour $j$ variant de 1 à $k$, alors la complexité du problème de la synthèse décroît de $2^{(\sum_{j=1}^{k} n_j)^\alpha}$ à $\sum_{j=1}^{k} 2^{n_j^\alpha}$, avec $\alpha$ une constante fixée. De tels résultats permettent d'améliorer le logiciel CROCUS, disponible à l'adresse suivante http://libresource.inria.fr//projects/crocus, que nous développons actuellement et qui trouve des quasi-interprétations à l'aide d'heuristiques.

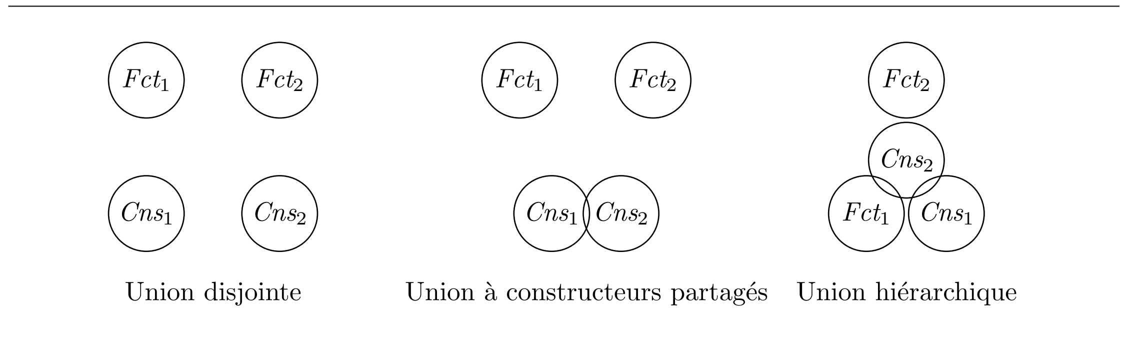


Fig. I.7: Unions de deux programmes $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ et $\langle Var_2, Cns_2, Fct_2, R_2\rangle$

### I.4.1 Union disjointe

Deux programmes $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ et $\langle Var_2, Cns_2, Fct_2, R_2\rangle$ sont disjoints si :

$$Fct_1 \cap Fct_2 = Cns_1 \cap Cns_2 = Fct_1 \cap Cns_2 = Fct_2 \cap Cns_1 = \emptyset$$

L'union disjointe de deux programmes $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ et $\langle Var_2, Cns_2, Fct_2, R_2\rangle$

disjoints est définie comme étant le programme suivant :

$$\langle Var_1, Cns_1, Fct_1, R_1\rangle \biguplus \langle Var_2, Cns_2, Fct_2, R_2\rangle = \langle Var_1 \cup Var_2, Cns_1 \cup Cns_2, Fct_1 \cup Fct_2, R_1 \cup R_2\rangle$$

Kurihara et Ohuchi [KO92] ont démontré que la terminaison est modulaire dans le cas de l'union disjointe. Nous en déduisons le résultat suivant :

**Proposition I.67.** *La propriété d'être ordonné par $\prec_{rpo}$ est modulaire relativement à l'union disjointe. En particulier, supposons que $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ est un programme ordonné par $\prec_{rpo}$ avec un statut $st_1$ et que $\langle Var_2, Cns_2, Fct_2, R_2\rangle$ est un programme ordonné par $\prec_{rpo}$ avec un statut $st_2$. Supposons que les programmes $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ et $\langle Var_2, Cns_2, Fct_2, R_2\rangle$ sont disjoints, alors $\langle Var_1, Cns_1, Fct_1, R_1\rangle \biguplus \langle Var_2, Cns_2, Fct_2, R_2\rangle$ est ordonné par $\prec_{rpo}$ avec un statut st défini par :*

$$st(\mathtt{f}) = st_i(\mathtt{f}) \qquad \mathtt{f} \in Fct_i \qquad i \in \{1, 2\}$$

Ainsi, si deux programmes sont ordonnés par $\prec_{rpo}$, où chaque symbole de fonction a un statut produit, la proposition ci-dessus garantit que leur union disjointe est aussi ordonnée par $\prec_{rpo}$ où chaque symbole a un statut produit.

À présent, nous nous intéressons au problème de la modularité des quasi-interprétations relativement à une union disjointe.

**Proposition I.68.** *La propriété d'avoir une quasi-interprétation est modulaire relativement à l'union disjointe. En d'autres termes, étant donnés deux programmes disjoints $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$, les programmes $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ admettent une quasi-interprétation si et seulement si $\boldsymbol{p}_1 \biguplus \boldsymbol{p}_2$ admet une quasi-interprétation.*

*Démonstration.* Étant donnés deux programmes $\mathbf{p}_1$ et $\mathbf{p}_2$, si $\mathbf{p}_1 \biguplus \mathbf{p}_2$ a une quasi-interprétation $(\!|-|\!)$, alors $(\!|-|\!)$ est aussi une quasi-interprétation de $\mathbf{p}_1$ et de $\mathbf{p}_2$. Réciproquement, supposons que $\mathbf{p}_1$ admette une quasi-interprétation $(\!|-|\!)_1$ et que $\mathbf{p}_2$ admette une quasi-interprétation $(\!|-|\!)_2$. Puisque les ensembles de symboles de fonction et de symboles de constructeur de chaque sous-programme sont disjoints deux à deux, pour tout symbole $b \in Cns_i \cup Fct_i$, on définit $(\!|b|\!)_{\mathbf{p}_1 \biguplus \mathbf{p}_2} = (\!|b|\!)_i$. Nous laissons le lecteur vérifier que l'assignation $(\!|-|\!)_{\mathbf{p}_1 \biguplus \mathbf{p}_2}$ ainsi définie est une quasi-interprétation pour $\mathbf{p}_1 \biguplus \mathbf{p}_2$. □

Dans la preuve qui précède, on peut remarquer que la quasi-interprétation $(\!|-|\!)_{\mathbf{p}_1 \biguplus \mathbf{p}_2}$ est additive si et seulement si $(\!|-|\!)_1$ et $(\!|-|\!)_2$ sont additives. De sorte que :

**Corollaire I.69.** *La propriété d'avoir une quasi-interprétation additive est modulaire relativement à l'union disjointe.*

Ce résultat, quoi qu'évident, nous donne une stratégie intéressante pour diminuer la complexité du problème de la synthèse de quasi-interprétations. Cette stratégie consiste à diviser un programme en sous-programmes ayant des ensembles de symboles disjoints, puis à itérer cette division autant que faire se peut. Si nous parvenons à diviser un programme en unions disjointes de $k$ sous-programmes ayant $n_j$ variables pour $j$ de 1 à $k$, alors la complexité de la synthèse de quasi-interprétation décroît de $2^{(\sum_{j=1}^{k} n_j)^\alpha}$ à $\sum_{j=1}^{k} 2^{n_j^\alpha}$, pour une constante arbitraire $\alpha$.

### I.4.2 Union à constructeurs partagés

Deux programmes $\langle \mathit{Var}_1, \mathit{Cns}_1, \mathit{Fct}_1, R_1 \rangle$ et $\langle \mathit{Var}_2, \mathit{Cns}_2, \mathit{Fct}_2, R_2 \rangle$ ont des constructeurs partagés si :

$$\mathit{Fct}_1 \cap \mathit{Fct}_2 = \mathit{Fct}_1 \cap \mathit{Cns}_2 = \mathit{Fct}_2 \cap \mathit{Cns}_1 = \emptyset$$

En d'autres termes, deux programmes ont des constructeurs partagés si leurs seuls symboles partagés sont des symboles de constructeur. L'union à constructeurs partagés de deux programmes $\langle \mathit{Var}_1, \mathit{Cns}_1, \mathit{Fct}_1, R_1 \rangle$ et $\langle \mathit{Var}_2, \mathit{Cns}_2, \mathit{Fct}_2, R_2 \rangle$ possédant des constructeurs partagés est définie comme le programme suivant :

$$\langle \mathit{Var}_1, \mathit{Cns}_1, \mathit{Fct}_1, R_1 \rangle \bigsqcup \langle \mathit{Var}_2, \mathit{Cns}_2, \mathit{Fct}_2, R_2 \rangle = \langle \mathit{Var}_1 \cup \mathit{Var}_2, \mathit{Cns}_1 \cup \mathit{Cns}_2, \mathit{Fct}_1 \cup \mathit{Fct}_2, R_1 \cup R_2 \rangle$$

La sémantique d'un tel programme est bien définie puisque la confluence est modulaire pour l'union à constructeurs partagés de systèmes de réécriture linéaires gauche [Rao95].

**Proposition I.70** (Modularité de $\prec_{rpo}$)**.** *Supposons que $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ soient deux programmes ordonnés par $\prec_{rpo}$ alors $\boldsymbol{p}_1 \bigsqcup \boldsymbol{p}_2$ est aussi ordonné par $\prec_{rpo}$, avec le même statut.*

Nous allons démontrer un résultat négatif concernant la modularité des quasi-interprétations dans le cas d'une union à constructeurs partagés :

**Proposition I.71.** *La propriété d'avoir une quasi-interprétation additive n'est pas modulaire relativement à l'union à constructeurs partagés.*

*Démonstration.* Nous démontrons cette proposition à l'aide d'un contre-exemple. À cet effet, considérons deux programmes $\mathbf{p}_0$ et $\mathbf{p}_1$. Les deux ensembles de symboles de constructeur correspondant $\mathit{Cns}_0$ et $\mathit{Cns}_1$ sont choisis comme étant égaux à $\{\mathbf{a}, \mathbf{b}\}$, $\mathit{Fct}_0 = \{\mathtt{f}_0\}$ et $\mathit{Fct}_1 = \{\mathtt{f}_1\}$. Enfin, les définitions des deux programmes sont les suivantes :

$$\begin{aligned}
\mathtt{f}_0(y) = \textbf{Case } y \textbf{ of } & \mathbf{a}(x) \to \mathtt{f}_0(\mathtt{f}_0(x)) \\
& \mathbf{b}(x) \to \mathbf{a}(\mathbf{a}(\mathtt{f}_0(x))) \\
\mathtt{f}_1(y) = \textbf{Case } y \textbf{ of } & \mathbf{b}(x) \to \mathtt{f}_1(\mathtt{f}_1(x)) \\
& \mathbf{a}(x) \to \mathbf{b}(\mathbf{b}(\mathtt{f}_1(x)))
\end{aligned}$$

$\mathbf{p}_0$ et $\mathbf{p}_1$ admettent les quasi-interprétations additives $(\!|-|\!)_0$ and $(\!|-|\!)_1$ définies par :

| | |
|---|---|
| $(\!\|\mathbf{a}\|\!)_0(X) = X+1$ | $(\!\|\mathbf{a}\|\!)_1(X) = X+2$ |
| $(\!\|\mathbf{b}\|\!)_0(X) = X+2$ | $(\!\|\mathbf{b}\|\!)_1(X) = X+1$ |
| $(\!\|\mathtt{f}_0\|\!)_0(X) = X$ | $(\!\|\mathtt{f}_1\|\!)_1(X) = X$ |

Nous allons raisonner par l'absurde en démontrant que $\mathbf{p}_0 \bigsqcup \mathbf{p}_1$ n'admet pas de quasi-interprétation additive. Supposons qu'il admette la quasi-interprétation additive $(\!|-|\!)$. Puisque $(\!|-|\!)$ est additive, on supposera que $(\!|\mathbf{a}|\!)(X) = X + k_\mathbf{a}$ et que $(\!|\mathbf{b}|\!)(X) = X + k_\mathbf{b}$, avec $k_\mathbf{a}, k_\mathbf{b} \geq 1$. Afin de simplifier la preuve, supposons que le polynôme $(\!|\mathtt{f}_0|\!)$ s'écrive sans utiliser l'opération max. Pour la première définition de $\mathbf{p}_0$, la quasi-interprétation $(\!|\mathtt{f}_0|\!)$ doit vérifier l'inégalité suivante :

$$(\!|\mathtt{f}_0|\!)(X + k_\mathbf{a}) \geq (\!|\mathtt{f}_0|\!)((\!|\mathtt{f}_0|\!)(X))$$

À présent, supposons que $(\!|\mathtt{f}_0|\!)(X)$ s'écrive de la forme $\alpha X^d + Q(X)$, avec $Q$ un polynôme de degré strictement inférieur à $d$ et avec une constante $\alpha \neq 0$. On constate que $(\!|\mathtt{f}_0|\!)(X + k_\mathbf{a})$ se réécrit en un polynôme de la forme $\alpha X^d + R(X)$, avec $R$ un polynôme de degré strictement inférieur à $d$, et que $(\!|\mathtt{f}_0|\!)((\!|\mathtt{f}_0|\!)(X))$ est égal à un polynôme de la forme $\alpha^2 X^{d^2} + S(X)$, où $S$ est un polynôme de degré strictement inférieur à $d^2$. Pour des valeurs de $X$ choisies suffisamment grandes, l'inégalité ci-dessus implique que $d \geq d^2$, ce qui implique que la constante $d$ est égale à 1. Maintenant, nous comparons les coefficients de plus haut degré de ces polynômes et on obtient, $\alpha \geq \alpha^2$. Ainsi, la constante $\alpha$ est, elle aussi, égale à 1 et, nous en concluons que $(\!|\mathtt{f}_0|\!)(X) = X + k$. Par symétrie, le même résultat est valable pour la quasi-interprétation de $\mathtt{f}_1$ qui s'écrit alors $(\!|\mathtt{f}_1|\!)(X) = X + k'$.

Les deux dernières définitions du programme impliquent les inégalités suivantes :

$$k_\mathbf{b} + k \geq 2k_\mathbf{a} + k$$
$$k_\mathbf{a} + k' \geq 2k_\mathbf{b} + k'$$

Par conséquent, $k_\mathbf{a} = k_\mathbf{b} = 0$, ce qui contredit le fait que $k_\mathbf{a}, k_\mathbf{b} \geq 1$. Pour conclure, l'union à constructeurs partagés des deux programmes n'admet pas de quasi-interprétation. $\square$

**Exemple 8** (Codage et décodage)**.** *Considérons le programme* $\boldsymbol{p} = \boldsymbol{p}_1 \bigsqcup \boldsymbol{p}_2$ *obtenu par union à constructeurs partagés des programmes* $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *définis ci-dessous :*

$$\begin{aligned}
\mathtt{code}_{\mathbf{1}}(z) &= \mathbf{Case}\ z\ \mathbf{of}\ \mathbf{0}(x) \to \mathbf{1}(\mathtt{code}_{\mathbf{1}}(x)) \\
&\qquad\qquad\qquad \mathbf{nil} \to \mathbf{nil} \\
\mathtt{shuffle}_{\mathbf{1}}(z,w) &= \mathbf{Case}\ z,w\ \mathbf{of}\ \mathbf{1}(x),\mathbf{1}(y) \to \mathbf{1}(\mathbf{1}(\mathtt{shuffle}_{\mathbf{1}}(x,y))) \\
&\qquad\qquad\qquad \mathbf{1}(x),\mathbf{0}(y) \to \mathbf{1}(\mathtt{shuffle}_{\mathbf{1}}(x,\mathtt{code}_{\mathbf{1}}(\mathbf{0}(y)))) \\
&\qquad\qquad\qquad \mathbf{nil},y \to \mathtt{code}_{\mathbf{1}}(y)
\end{aligned}$$

$$\begin{aligned}
\mathtt{code}_{\mathbf{0}}(z) &= \mathbf{Case}\ z\ \mathbf{of}\ \mathbf{1}(x) \to \mathtt{f}(\mathbf{0}(\mathbf{0}(\mathtt{code}_{\mathbf{0}}(x)))) \\
&\qquad\qquad\qquad \mathbf{nil} \to \mathbf{nil} \\
\mathtt{f}(z) &= \mathbf{Case}\ z\ \mathbf{of}\ \mathbf{0}(\mathbf{0}(x)) \to \mathbf{0}(x) \\
\mathtt{shuffle}_{\mathbf{0}}(z,w) &= \mathbf{Case}\ z,w\ \mathbf{of}\ \mathbf{0}(x),\mathbf{0}(y) \to \mathbf{0}(\mathbf{0}(\mathtt{shuffle}_{\mathbf{0}}(x,y))) \\
&\qquad\qquad\qquad \mathbf{0}(x),\mathbf{1}(y) \to \mathbf{0}(\mathtt{shuffle}_{\mathbf{0}}(x,\mathtt{code}_{\mathbf{0}}(\mathbf{1}(y)))) \\
&\qquad\qquad\qquad \mathbf{nil},y \to \mathtt{code}_{\mathbf{0}}(y)
\end{aligned}$$

*Les deux symboles de fonction* $\mathtt{code}_{\mathbf{0}}$ *et* $\mathtt{code}_{\mathbf{1}}$ *calculent des fonctions réciproques sur les mots binaires* $\{0,1\}^*$. *En effet, on a* $[\![\mathtt{code}_{\mathbf{0}}(\mathtt{code}_{\mathbf{1}})(\mathbf{0}^n(\mathbf{nil}))]\!] = \mathbf{0}^n(\mathbf{nil})$ *et inversement. De plus, les programmes* $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *admettent les quasi-interprétations respectives* $(\!|-|\!)_1$ *et* $(\!|-|\!)_2$ *définies par :*

$$\begin{aligned}
(\!|\mathbf{nil}|\!)_1 &= 0 \\
(\!|\mathbf{0}|\!)_1(X) &= X+1 \\
(\!|\mathbf{1}|\!)_1(X) &= X+1 \\
(\!|\mathtt{code}_{\mathbf{1}}|\!)_1(X) &= X \\
(\!|\mathtt{shuffle}_{\mathbf{1}}|\!)_1(X,Y) &= X+Y
\end{aligned}
\qquad\Bigg|\qquad
\begin{aligned}
(\!|\mathbf{nil}|\!)_2 &= 0 \\
(\!|\mathbf{0}|\!)_2(X) &= X+1 \\
(\!|\mathbf{1}|\!)_2(X) &= X+2 \\
(\!|\mathtt{code}_{\mathbf{0}}|\!)_2(X) &= X \\
(\!|\mathtt{shuffle}_{\mathbf{0}}|\!)_2(X,Y) &= X+Y \\
(\!|\mathtt{f}|\!)_2(X) &= X
\end{aligned}$$

*Raisonnons par l'absurde en supposant que* $\boldsymbol{p}_1 \bigsqcup \boldsymbol{p}_2$ *admet une quasi-interprétation additive* $(\!|-|\!)$ *telle que* $(\!|\mathbf{1}|\!)(X) = X + k_{\mathbf{1}}$ *et* $(\!|\mathbf{0}|\!)(X) = X + k_{\mathbf{0}}$, *avec* $k_{\mathbf{1}}, k_{\mathbf{0}} \geq 1$. *Les définitions de* $\mathtt{shuffle}_{\mathbf{0}}$ *et de* $\mathtt{shuffle}_{\mathbf{1}}$ *de la forme :*

$$\mathtt{shuffle}_{\boldsymbol{i}}(\boldsymbol{i}(x),\boldsymbol{j}(y)) \to \boldsymbol{i}(\mathtt{shuffle}_{\boldsymbol{i}}(x,\mathtt{code}_{\boldsymbol{i}}(\boldsymbol{j}(y))))\,\textit{avec}\ \boldsymbol{i},\boldsymbol{j} \in \{\mathbf{0},\mathbf{1}\},\ \boldsymbol{i}+\boldsymbol{j} = \mathbf{1}$$

*forcent la quasi-interprétation de* $\mathtt{code}_{\mathbf{0}}$ *et celle de* $\mathtt{code}_{\mathbf{1}}$ *à être au plus égales à* $X + d$

*pour une constante d arbitraire. En effet, par définition des quasi-interprétations, on a :*

$$\begin{aligned}
(\!|\mathtt{shuffle}_{\boldsymbol{i}}(\boldsymbol{i}(x), \boldsymbol{j}(y))|\!) &= (\!|\mathtt{shuffle}_{\boldsymbol{i}}|\!)(X + k_{\boldsymbol{i}}, Y + k_{\boldsymbol{j}}) \\
&\geq (\!|\boldsymbol{i}(\mathtt{shuffle}_{\boldsymbol{i}}(x, \mathtt{code}_{\boldsymbol{i}}(\boldsymbol{j}(y))))|\!) \\
&= k_{\boldsymbol{i}} + (\!|\mathtt{shuffle}_{\boldsymbol{i}}|\!)(X, (\!|\mathtt{code}_{\boldsymbol{i}}|\!)(Y + k_{\boldsymbol{j}}))
\end{aligned}$$

*Il en résulte que le programme n'admet pas de quasi-interprétation additive et polynomiale $(\!|\mathtt{shuffle}_{\boldsymbol{i}}|\!)$ si $(\!|\mathtt{code}_{\boldsymbol{i}}|\!)(X)$ est supérieur à $X + d$, avec une constante $d$ fixée. Les premières définitions de $\boldsymbol{p}_1$ et de $\boldsymbol{p}_2$ fournissent les inégalités suivantes :*

$$\begin{aligned}
(\!|\mathtt{code}_{\mathbf{1}}(\mathbf{0}(x))|\!) &= X + d + k_{\mathbf{0}} \geq X + d + k_{\mathbf{1}} = (\!|\mathbf{1}(\mathtt{code}_{\mathbf{1}}(x))|\!) && \textit{Pour } \boldsymbol{p}_1 \\
(\!|\mathtt{code}_{\mathbf{0}}(\mathbf{1}(x))|\!) &= X + d + k_{\mathbf{1}} \geq (\!|\mathtt{f}(\mathbf{0}(\mathbf{0}(\mathtt{code}_{\mathbf{0}}(x))))|\!) && \textit{Pour } \boldsymbol{p}_2 \\
&\geq X + d + 2 \times k_{\mathbf{0}} && \textit{Sous-terme}
\end{aligned}$$

*La combinaison des deux inégalités implique que $k_{\mathbf{0}} \geq 2 \times k_{\mathbf{0}}$, ce qui est incompatible avec la contrainte $k_{\mathbf{0}} \geq 1$. Ainsi, $\boldsymbol{p}$ n'admet aucune quasi-interprétation. Finalement, il nous reste à remarquer que les deux sous-programmes terminent par $\prec_{rpo}$ avec un statut lexicographique.*

Cependant, les bornes de complexité demeurent polynomiales même si l'union à constructeurs partagés n'admet pas de quasi-interprétation. Nous commençons par établir que le lemme fondamental I.23 est encore valide :

**Théorème I.72.** *Étant donnés deux programmes $\boldsymbol{p}_1 = \langle Var_1, Cns_1, Fct_1, R_1 \rangle$ et $\boldsymbol{p}_2 = \langle Var_2, Cns_2, Fct_2, R_2 \rangle$ admettant des quasi-interprétations additives, il existe un polynôme $P$ tel que pour toute expression $t$ de $\boldsymbol{p}_1 \bigsqcup \boldsymbol{p}_2$ ayant $n$ variables $x_1, \cdots, x_n$, et pour toute substitution $\sigma$ vérifiant $x_i\sigma = v_i$, on ait :*

$$|[\![t\sigma]\!]| \leq P^{|t|}(\max_{i=1..n} |v_i|)$$

*Démonstration.* La preuve est une conséquence directe du lemme I.23. Observons tout d'abord qu'il n'y a aucun appel entre deux programmes $\mathbf{p}_1$ et $\mathbf{p}_2$ d'une union à constructeurs partagés $\mathbf{p}_1 \bigsqcup \mathbf{p}_2$. Le lemme fondamental est valide pour chacun des programmes $\mathbf{p}_1$ et $\mathbf{p}_2$ avec des polynômes respectifs $P_1$ et $P_2$. Définissons $P = \max(P_1, P_2)$. Pour toute séquence de valeurs $v_1, \cdots, v_n \in \mathcal{T}(Cns_1 \cup Cns_2)$, l'expression $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{*} v$, avec $\mathtt{f} \in Fct_j$, est évaluée en utilisant uniquement des définitions de $\langle Var_j, Cns_j, Fct_j, R_j \rangle$. On applique donc le lemme fondamental I.23, obtenant l'inégalité suivante :

$$|v| \leq P_j(\max_{i=1..n} |v_i|) \leq P(\max_{i=1..n} |v_i|)$$

Il suffit désormais de terminer la preuve par une induction structurelle sur un terme $t$. Le cas $t = x$ est trivial. Pour $t = \mathtt{g}(t_1, \cdots, t_n)$, on combine l'inégalité précédente avec le fait que la composition de deux fonctions de **Max-Poly** reste polynomialement bornée. □

Puisque $\prec_{rpo}$ est modulaire dans le cas d'une union à constructeurs partagés, le théorème I.72 implique que :

**Corollaire I.73** (TEMPS et ESPACE pour l'union à constructeurs partagés)**.**

- *L'ensemble des fonctions calculées par une union à constructeurs partagés de deux programmes additifs ordonnés par $\prec_{rpo}$ où chaque symbole de fonction possède un statut produit est exactement l'ensemble des fonctions calculables en temps polynomial.*
- *L'ensemble des fonctions calculées par une union à constructeurs partagés de deux programmes additifs ordonnés par $\prec_{rpo}$ est exactement l'ensemble des fonctions calculables en espace polynomial.*

Le théorème I.72 permet d'analyser plus de programmes comme le contre-exemple fournit dans l'exemple 8. En effet, il existe de nombreux algorithmes basés sur des procédures de codage et de décodage qui n'étaient pas capturés par le théorème I.36 et qui sont désormais capturés en divisant le programme en deux sous-programmes à constructeurs partagés. Tout programme peut être partagé entre une union à constructeurs partagés de lui-même et d'un programme sans symbole de fonction mais avec le même ensemble de symboles de constructeur. La caractérisation en espace établie ci-dessus est donc intensionnellement plus puissante que la caractérisation du théorème I.36. La stratégie évoquée dans la section précédente pour la synthèse de quasi-interprétation se trouve elle-même améliorée puisque l'union disjointe de deux programmes n'est finalement qu'un cas particulier d'union à constructeurs partagés. On peut donc synthétiser les quasi-interprétations de manière plus flexible en divisant un programme donné en une union à constructeurs partagés de deux sous-programmes, puis itérer cette division autant que possible.

### I.4.3 Union hiérarchique

Deux programmes $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ et $\langle Var_2, Cns_2, Fct_2, R_2\rangle$ sont hiérarchiques si

$$Fct_1 \cap Fct_2 = Fct_2 \cap Cns_1 = \emptyset \text{ et } Cns_1 \cap Cns_2 \neq \emptyset \text{ et } Fct_1 \cap Cns_2 \neq \emptyset$$

où les symboles de $Fct_1$ n'apparaissent pas dans les motifs de $R_2$. L'union hiérarchique de deux programmes hiérarchiques se définit comme le programme suivant :

$$\begin{aligned}&\langle Var_1, Cns_1, Fct_1, R_1\rangle \ll \langle Var_2, Cns_2, Fct_2, R_2\rangle\\ &= \langle Var_1 \cup Var_2, Cns_1 \cup Cns_2 - Fct_1, Fct_1 \cup Fct_2, R_1 \cup R_2\rangle\end{aligned}$$

Tout d'abord, remarquons que l'opérateur d'union hiérarchique $\ll$ n'est plus un opérateur commutatif contrairement à ceux des unions disjointes ou à constructeurs partagés. En effet, le programme $\langle Var_2, Cns_2, Fct_2, R_2\rangle$ appelle des symboles de fonction du programme $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ et la réciproque n'est pas vrai. Autrement dit, l'union hiérarchique de deux programmes $\mathbf{p}_1 \ll \mathbf{p}_2$ correspond à un programme $\mathbf{p}_2$ qui peut charger et exécuter des librairies de $\mathbf{p}_1$.

L'hypothèse que les motifs de $R_2$ sont à valeur sur les symboles de $Cns_2 - Fct_1$ implique qu'il n'existe aucune paire critique. En conséquence, la confluence est une propriété modulaire des unions hiérarchiques considérées et la sémantique de nos programmes est ainsi clairement définie.

**Proposition I.74** (Modularité de $\prec_{rpo}$)**.** *Soient $\langle Var_1, Cns_1, Fct_1, R_1\rangle$ un programme ordonné par $\prec_{rpo}$ avec le statut $st_1$ et la précédence $\geq_{\mathcal{F}_1}$ et $\langle Var_2, Cns_2, Fct_2, R_2\rangle$ un programme ordonné par $\prec_{rpo}$ avec le statut $st_2$ et la précédence $\geq_{\mathcal{F}_2}$, alors le programme $\langle Var_1, Cns_1, Fct_1, R_1\rangle \ll \langle Var_2, Cns_2, Fct_2, R_2\rangle$ est ordonné par $\prec_{rpo}$ avec un statut st et une précédence $\preceq_{\mathcal{F}_1\cup\mathcal{F}_2}$ définis par :*

$$\begin{array}{lll} st(\mathtt{f}) = st_i(\mathtt{f}) & \mathtt{f} \in Fct_i & i \in \{1,2\}\\ \mathtt{g} \preceq_{\mathcal{F}_1\cup\mathcal{F}_2} \mathtt{f} & si\ \mathtt{g} \preceq_{\mathcal{F}_i} \mathtt{f} & i \in \{1,2\}\\ \mathtt{g} \prec_{\mathcal{F}_1\cup\mathcal{F}_2} \mathtt{f} & si\ \mathtt{f} \in Fct_2\ \ et\ \mathtt{g} \in Fct_1 \cap Cns_2 & \end{array}$$

Puisque l'union à constructeurs partagés est un cas particulier d'union hiérarchique (en choisissant $Fct_1 \cap Cns_2 = \emptyset$), nous obtenons le résultat suivant :

**Proposition I.75.** *La propriété d'admettre une quasi-interprétation additive n'est pas modulaire relativement à l'union hiérarchique.*

De plus, contrairement à l'union à constructeurs partagés, le lemme fondamental n'est plus valide dans le cas d'une union hiérarchique. Voici un contre-exemple :

**Exemple 9.** *Les programmes $\boldsymbol{p}_1$ :*

$$\begin{aligned}\mathtt{double}(y) = \mathbf{Case}\ y\ \mathbf{of}\ &\mathbf{S}(x) \to \mathbf{S}(\mathbf{S}(\mathtt{double}(x)))\\ &\mathbf{0} \to \mathbf{0}\end{aligned}$$

*et* $\boldsymbol{p}_2$ *:*

$$\begin{aligned}\exp(y) = \textbf{Case } y \textbf{ of } & \mathbf{S}(x) \to \texttt{double}(\exp(x)) \\ & \mathbf{0} \to \mathbf{S}(\mathbf{0})\end{aligned}$$

*sont ordonnés par* $\prec_{rpo}$ *avec un statut produit et admettent les quasi-interprétations additives suivantes :*

| | |
|---|---|
| $(\!\|\mathbf{0}\|\!)_1 = 0$ | $(\!\|\mathbf{0}\|\!)_2 = 0$ |
| $(\!\|\mathbf{S}\|\!)_1(X) = X + 1$ | $(\!\|\mathbf{S}\|\!)_2(X) = (\!\|\texttt{double}\|\!)_2(X) = X + 1$ |
| $(\!\|\texttt{double}\|\!)_1(X) = 2 \times X$ | $(\!\|\exp\|\!)_2(X) = X + 1$ |

`double` *peut être assimilé à un symbole de constructeur dont la quasi-interprétation est additive dans* $\boldsymbol{p}_2$ *alors qu'il représente un symbole de fonction dont la quasi-interprétation est affine dans* $\boldsymbol{p}_1$*. Cependant l'union hiérarchique de ces deux programmes calcule une fonction exponentielle. Cette exponentielle provient des différents types de polynômes attribués au symbole* `double`*.*

Dès à présent, nous allons établir des restrictions de manière à préserver le lemme fondamental. Afin d'éviter le contre-exemple ci-dessus, nous posons des restrictions sur la forme des polynômes admissibles pour les quasi-interprétations de symboles partagés dans un *critère de préservation*. Ce critère a été introduit dans [BMP07], où il est appelé Kind-preserving, et s'inspire de la distinction faite dans [BCMT99] entre des polynômes qu'ils soient linéaires, affines ou autres.

Dans cette optique, on définit un *polynôme honnête* comme étant un polynôme au sens strict (c'est-à-dire sans l'usage de l'opération max) et dont les coefficients sont tous supérieurs ou égaux à 1. Par extension, une quasi-interprétation $(\!|-|\!)$ est dite honnête si $(\!|b|\!)$ est honnête pour tout symbole $b$. Les polynômes honnêtes sont très utilisés en pratique à cause de la propriété sous-terme.

Étant donnés $n$ variables $X_1, \cdots, X_n$ et $n$ entiers naturels $a_1, \cdots, a_n$, on définit un monôme $m$ comme étant un polynôme d'un seul terme pouvant s'écrire $m(X_1, \cdots, X_n) = X_1^{a_1} \times \ldots \times X_n^{a_n}$ où il existe au moins une valeur de $j$ telle que $a_j \neq 0$. Étant donnés un monôme $m$ et un polynôme $P$, on définit la relation « $m$ est un monôme de $P$ »de la manière suivante : $m \sqsubseteq P$ si et seulement si $P = \sum_{j=1}^{k} \alpha_j \times m_j$, avec des constantes $\alpha_j$ et des monômes $m_j$ distincts deux à deux, et s'il existe $i \in \{1, k\}$ tel que $m_i = m$ et $\alpha_i \neq 0$. Dans ce cas, le coefficient $\alpha_i$, noté $\texttt{coef}_P(m)$, est défini comme le coefficient multiplicatif associé à $m$ dans $P$.

**Définition I.76.** *Considérons* $\langle Var_1, Cns_1, Fct_1, R_1\rangle \ll \langle Var_2, Cns_2, Fct_2, R_2\rangle$ *l'union hiérarchique de deux programmes possédant des quasi-interprétations respectives* $(\!|-|\!)_1$ *et* $(\!|-|\!)_2$*. On dit que* $(\!|-|\!)_1$ *et* $(\!|-|\!)_2$ *sont* ***préservatrices*** *si* $\forall b \in Cns_2 \cap Fct_1$ *:*

1. *$(\!|b|\!)_1$ et $(\!|b|\!)_2$ sont des polynômes honnêtes.*
2. *$\forall m$, $m \sqsubseteq (\!|b|\!)_1 \Leftrightarrow m \sqsubseteq (\!|b|\!)_2$*
3. *$\forall m$, $\texttt{coef}_{(\!|b|\!)_2}(m) = 1 \Leftrightarrow \texttt{coef}_{(\!|b|\!)_1}(m) = 1$*

*Deux quasi-interprétations $(\!|-|\!)_1$ et $(\!|-|\!)_2$ sont additivement préservatrices si les deux conditions ci-dessous sont remplies :*
- *$(\!|b|\!)_1$ est additive pour tout $b \in Cns_1$,*
- *$(\!|b|\!)_2$ est additive pour tout $b \in Cns_2 - Fct_1$.*

Remarquons que $(\!|b|\!)_2$ n'est pas nécessairement additive pour les symboles $b \in Cns_2 \cap Fct_1$.

**Exemple 10.** *Les quasi-interprétations $(\!|-|\!)_1$ et $(\!|-|\!)_2$ de l'exemple 9 ne sont pas préservatrices puisque l'on a $(\!|\texttt{double}|\!)_1(X) = 2 \times X$ et $(\!|\texttt{double}|\!)_2(X) = X + 1$.*

Par conséquent, une restriction naturelle pour préserver le lemme fondamental pourrait être de forcer les quasi-interprétations d'une union hiérarchique à être additivement préservatrices. Cependant, une telle restriction n'est pas suffisante comme l'illustre le contre-exemple ci-dessous :

**Exemple 11.** *On considère les programmes $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ :*

$$\texttt{g}(t) = \mathbf{S}(\mathbf{S}(t))$$

$$\begin{aligned}\texttt{f}(u,v,w) = \textbf{Case } u,v,w \textbf{ of } \mathbf{S}(x),\mathbf{0},t &\to \texttt{f}(x,t,t)\\ x,\mathbf{S}(z),t &\to \texttt{f}(x,z,\texttt{g}(t))\\ \mathbf{0},\mathbf{0},t &\to t\end{aligned}$$

*Leur union hiérarchique $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ calcule une fonction exponentielle. En utilisant la notation $\underline{n}$ pour $\underbrace{\mathbf{S}(\ldots\mathbf{S}(\mathbf{0})\ldots)}_{n \text{ fois } \mathbf{S}}$, on a $[\![\texttt{f}]\!](\underline{n},\underline{m},\underline{p}) = \underline{3^n \times (2 \times m + p)}$.*

*$\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ sont ordonnés par $\prec_{rpo}$ avec un statut lexicographique et admettent les quasi-interprétations additivement préservatrices suivantes :*

$$\begin{aligned}(\!|\mathbf{S}|\!)_1(X) &= X + 1\\ (\!|\texttt{g}|\!)_1(X) &= X + 2\end{aligned}$$

$$\begin{aligned}(\!|\mathbf{0}|\!)_2 &= 0\\ (\!|\mathbf{S}|\!)_2(X) &= (\!|\texttt{g}|\!)_2(X) = X + 1\\ (\!|\texttt{f}|\!)_2(X,Y,Z) &= \max(X,Y,Z)\end{aligned}$$

Dans ce contre-exemple, $(\!|\texttt{f}|\!)_2$ n'est pas un polynôme honnête puisque `f` n'est pas un symbole partagé. Le problème de ce contre-exemple est directement lié au fait que le nombre d'alternances entre des définitions des deux programmes utilisées durant l'évaluation dépend directement de la taille des entrées. Ainsi, un critère suffisant permettant de surmonter cet inconvénient est de borner ce nombre d'alternances par une constante.

Dans cette perspective, nous allons définir des restrictions syntactiques sur les programmes considérés en utilisant la notion de programme plat (ou flat program) introduite par Dershowitz dans [Der94], où elle était utilisée afin d'assurer la modularité de la complétude de l'union hiérarchique.

**Définition I.77.** *L'union hiérarchique* $\langle Var_1, Cns_1, Fct_1, R_1\rangle \ll \langle Var_2, Cns_2, Fct_2, R_2\rangle$ *est dite* stratifiée *si :*

- *Pour toute expression* $e$ *activée par* $\mathtt{f}(p_1, \cdots, p_n)$ *dans* $R_2$*, on a : Pour tout sous-terme* $\mathtt{g}(e_1, \cdots, e_n)$ *de* $e$ *vérifiant* $\mathtt{g} \approx_{Fct_2} \mathtt{f}$*, aucun symbole de fonction de* $Cns_2 \cap Fct_1$ *n'apparaît dans les arguments* $e_1, \cdots, e_n$ *de* $\mathtt{g}$*.*
- *Le programme* $\langle Var_2, Cns_2, Fct_2, R_2\rangle$ *est plat : Pour toute expression* $e$ *activée par* $\mathtt{f}(p_1, \cdots, p_n)$ *de* $R_2$*,* $e$ *ne possède pas de symboles de fonction imbriqués. C'est à dire,* $e$ *est une expression sans aucune composition de symboles de fonction dans* $Fct_2$*.*

Les exemples développés en section I.5.3 sont stratifiés et montrent que cette restriction capture tout de même un grand nombre d'algorithmes. Cependant, le programme de l'exemple 11 n'est pas une union stratifiée puisque la fonction $\mathtt{g}$ appartenant au programme $\mathbf{p}_1$ est utilisé en tant qu'argument d'une définition de $\mathtt{g}$ qui appartient au programme $\mathbf{p}_2$.

**Définition I.78** (Alternances)**.** *Considérons l'union hiérarchique de deux programmes* $\langle Var_1, Cns_1, Fct_1, R_1\rangle \ll \langle Var_2, Cns_2, Fct_2, R_2\rangle$ *et une expression* $t$*. On dit que* $t$ *est évaluée en utilisant* $k$ *alternances entre les définitions de* $R_1$ *et celles de* $R_2$ *s'il existe des expressions* $u_1, \cdots, u_k, v_1, \cdots, v_k$ *telles que :*

$$t \xrightarrow{*}_2 u_1 \xrightarrow{*}_1 v_1 \ldots \xrightarrow{*}_2 u_k \xrightarrow{*}_1 v_k$$

*où* $v_k$ *est une forme normale et* $\xrightarrow{*}_i$ *dénote la relation de réécriture induite par les définitions de* $R_i$*.*

On commence par établir qu'une union stratifiée peut être évaluée suivant une stratégie d'évaluation particulière en utilisant un nombre constant d'alternances :

**Lemme I.79.** *Considérons l'union stratifiée* $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ *de* $\boldsymbol{p}_1 = \langle Var_1, Cns_1, Fct_1, R_1\rangle$ *et de* $\boldsymbol{p}_2 = \langle Var_2, Cns_2, Fct_2, R_2\rangle$*. Pour tout symbole de fonction* $\mathtt{f}$ *d'arité* $n$ *et pour toute séquence de valeurs* $v_1, \cdots, v_n$*,* $\mathtt{f}(v_1, \cdots, v_n)$ *peut être évalué en utilisant un nombre constant d'alternances entre les définitions de* $R_1$ *et celles de* $R_2$*, i.e. le nombre d'alternance ne dépend pas des* $v_i$*.*

*Démonstration.* Étant donné un programme $\langle Var, Cns, Fct, \mathcal{R}\rangle$, on définit d'abord un rang sur les symboles de fonction de *Fct* à valeur sur les entiers naturels $\mathbb{N}$ qui satisfait :

$$\begin{aligned}
&\mathtt{rk}_{Fct}(\mathtt{f}) = 0 && \text{si } \nexists \mathtt{g} \in Fct, \text{ tel que } \mathtt{f} >_{Fct} \mathtt{g} \\
&\mathtt{rk}_{Fct}(\mathtt{f}) = \max_{\mathtt{g} \in Fct}(\mathtt{rk}_{Fct}(\mathtt{g})) + 1 && \text{si } \forall \mathtt{g} \in Fct,\ \mathtt{f} >_{Fct} \mathtt{g} \\
&\mathtt{rk}_{Fct}(\mathtt{f}) = \mathtt{rk}_{Fct}(\mathtt{g}) && \text{si } \mathtt{f} \approx_{Fct} \mathtt{g}
\end{aligned}$$

Nous allons démontrer ce lemme en utilisant une stratégie d'évaluation particulière. Puisque la confluence est une propriété modulaire de l'union hiérarchique des programmes considérés, comme nous l'avons mentionné en début de section, on peut évaluer un programme d'une telle manière.

- Si $\mathtt{f} \in Fct_1$ alors il n'y a aucune alternance et le résultat en découle directement.
- À présent, si $\mathtt{f} \in Fct_2$, nous allons démontrer par induction sur le rang qu'un symbole de fonction $\mathtt{f}$ de $Fct_2$ de rang $k$ peut être évalué en utilisant au plus $k+1$ alternances :
  - Si $\mathtt{rk}_{Fct_2}(\mathtt{f}) = 0$ alors tout symbole de fonction $\mathtt{g}$ apparaissant dans le membre droit d'une définition du symbole $\mathtt{f}$ est un symbole de fonction équivalent pour la précédence $\geq_{Fct_2}$. Par définition de l'union stratifiée, l'évaluation de $\mathtt{f}$ peut être effectuée en utilisant seulement des définitions de $R_2$. On évalue dans un premier temps tous ces appels récursifs. À la fin, soit une erreur advient, soit on obtient une valeur dans $\mathcal{T}(Cns_1 \cup Cns_2)$. Puisque tous les symboles de fonction de $Fct_2$ ont été évalués, on finit par évaluer les symboles de fonction de $Fct_1$ en appliquant seulement des définitions dans $R_1$. Finalement, l'évaluation a été effectuée en utilisant une unique alternance.
  - Supposons que si $\mathtt{rk}_{Fct_2}(\mathtt{g}) \leq n$, alors l'évaluation de $\mathtt{g}(u_1, \cdots, u_k)$ puisse être effectuée en utilisant au plus $n+1$ alternances et considérons le symbole de fonction $\mathtt{f}$ tel que $\mathtt{rk}_{Fct_2}(\mathtt{f}) = n+1$. Puisque l'union est stratifiée, tout appel récursif de la forme $\mathtt{h}(v_1, \cdots, v_m)$ avec $v_1, \cdots, v_m$ valeurs et $\mathtt{h} \approx_{Fct_2} \mathtt{f}$ peut être calculé en utilisant uniquement des définitions de $R_2$. Le fait que le programme $\mathbf{p}_2$ soit plat garantit que les symboles de fonction de $Fct_2$ ne sont pas composés lors des appels récursifs. En effet, une composition de symboles de fonction de $Fct_2$ apparaissant lors d'un appel récursif peut engendrer un nombre non borné d'alternances. Nous pouvons donc évaluer tout appel récursif d'un symbole de fonction $\mathtt{h}$ tel que $\mathtt{h} \approx_{Fct_2} \mathtt{f}$ en utilisant uniquement des définitions de $R_2$. Il reste alors à évaluer les symboles de fonction de rang strictement inférieur à $n+1$. On commence par évaluer leur arguments. Puisque le programme est plat, cette évaluation est effectuée en utilisant uniquement des définitions de $R_1$, de telle sorte que nous

créons une alternance. On applique l'hypothèse d'induction, ajoutant $n+1$ alternances supplémentaires, en évaluant tous ces symboles en parallèle, ce qui est possible puisqu'il n'y a aucune composition, et en éliminant tous les symboles de fonction restant dans $Fct_2$. Il ne reste ensuite que des symboles de fonction dans $Fct_1$ que l'on évalue en utilisant uniquement des définitions de $R_1$. Cette dernière évaluation n'ajoute aucune alternance supplémentaire. Finalement, le symbole de fonction de rang $n+1$ est évalué à l'aide de $n+2$ alternances entre les définitions de $\mathbf{p}_1$ et celles de $\mathbf{p}_2$.

La notion de rang introduite étant bornée par la taille du programme, on obtient le résultat requis. □

**Définition I.80** (Extension)**.** *Étant donnée l'union hiérarchique de deux programmes $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ possédant des quasi-interprétations respectives $(\!|-|\!)_1$ et $(\!|-|\!)_2$, on définit l'extension de la quasi-interprétation $(\!|-|\!)_1$ (resp. $(\!|-|\!)_2$), que nous notons aussi $(\!|-|\!)_1$ (resp. $(\!|-|\!)_2$), par $\forall b \in Cns_2 \cup Fct_2 \setminus (Cns_1 \cup Fct_1)$ (resp. $Cns_1 \cup Fct_1 \setminus (Cns_2 \cup Fct_2)$) $(\!|b|\!)_1 =_{def} (\!|b|\!)_2$ (resp. $(\!|b|\!)_2 =_{def} (\!|b|\!)_1$).*

*Les extensions de $(\!|-|\!)_1$ et de $(\!|-|\!)_2$ sont définies sur toute expression de $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$. De plus, si $(\!|-|\!)_1$ et $(\!|-|\!)_2$ sont des quasi-interprétations additivement préservatrices alors leurs extensions le sont aussi.*

**Lemme I.81.** *Soit l'union hiérarchique de deux programmes $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ admettant des quasi-interprétations préservatrices $(\!|-|\!)_1$ et $(\!|-|\!)_2$, alors il existe deux polynômes $P$ et $R$ tels que pour toute expression $w$ de $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$, les extensions de $(\!|-|\!)_1$ et $(\!|-|\!)_2$ satisfont :*

$$(\!|w|\!)_1^* \leq P((\!|w|\!)_2^*)$$
$$(\!|w|\!)_2^* \leq R((\!|w|\!)_1^*)$$

*Démonstration.* On démontre ce résultat en construisant le polynôme $P$, le résultat en découle par symétrie de la condition sur les quasi-interprétations préservatrices pour les extensions de $(\!|-|\!)_1$ et de $(\!|-|\!)_2$. Définissons $\alpha$ comme le plus petit coefficient multiplicatif strictement plus grand que 1 des polynômes $(\!|b|\!)_2$ (si un tel coefficient n'existe pas, alors la définition I.76 implique que les quasi-interprétations sont identiques) et définissons $\beta$ comme étant le plus grand coefficient multiplicatif et additif des polynômes $(\!|b|\!)_1$, pour tout symbole $b$. À présent, on définit quatre assignations $(\!|-|\!)_\alpha$, $(\!|-|\!)_{\alpha=1}$, $(\!|-|\!)_\beta$ et $(\!|-|\!)_{\beta=1}$ par :

- $(\!|-|\!)_\alpha$ est définie à partir de $(\!|-|\!)_2$ en remplaçant tout coefficient multiplicatif distinct de 1 par $\alpha$ et tout coefficient additif par 1.
- $(\!|-|\!)_{\alpha=1}$ est définie à partir de $(\!|-|\!)_\alpha$ en remplaçant tout coefficient multiplicatif distinct de 1 par 1.

- $(\!|-|\!)_\beta$ est définie à partir de $(\!|-|\!)_1$ en remplaçant tout coefficient multiplicatif et additif distinct de 1 par $\beta$.
- $(\!|-|\!)_{\beta=1}$ est définie à partir de $(\!|-|\!)_\beta$ en remplaçant tout coefficient multiplicatif et additif distinct de 1 par 1.

Intuitivement, $(\!|-|\!)_\alpha$ et $(\!|-|\!)_\beta$ représentent respectivement une borne inférieure de $(\!|-|\!)_2$ et une borne supérieure de $(\!|-|\!)_1$. On peut démontrer par induction structurelle que pour toute expression $w$, on a :

$$(\!|w|\!)^*_\alpha \leq (\!|w|\!)^*_2 \qquad \text{Par définition de } (\!|-|\!)^*_\alpha \tag{I.2}$$

$$(\!|w|\!)^*_1 \leq (\!|w|\!)^*_\beta \qquad \text{Par définition de } (\!|-|\!)^*_\beta \tag{I.3}$$

$$(\!|w|\!)^*_{\beta=1} = (\!|w|\!)^*_{\alpha=1} \qquad \text{Condition 2 de la définition I.76} \tag{I.4}$$

Dans la suite de cette preuve, nous allons procéder à un changement de point de vue en considérant $\alpha$ et $\beta$ comme des variables. Pour toute expression $w$, $(\!|w|\!)^*_\alpha$ et $(\!|w|\!)^*_\beta$ peuvent alors être vus comme des polynômes en $\alpha$ et, respectivement, en $\beta$ ; et ceci même si le degré de ces polynômes peut dépendre de la taille de l'expression $w$. Supposons que $(\!|w|\!)^*_\beta$ est un polynôme de degré $d$ en $\beta$. Puisque $\beta \geq 1$, par définition d'un polynôme honnête, pour tout $k \leq d$, $\beta^k \leq \beta^d$. Si on écrit $(\!|w|\!)^*_\beta$ sous la forme suivante $\sum_{i=1}^{d} \gamma_i \beta^i$, avec des constantes $\gamma_i$, on obtient :

$$(\!|w|\!)^*_\beta = \sum_{i=1}^{d} \gamma_i \beta^i \tag{I.5}$$

$$\leq (\sum_{i=1}^{d} \gamma_i) \times \beta^d \qquad \text{Puisque } \beta \geq 1 \tag{I.6}$$

$$= (\!|w|\!)^*_{\beta=1} \times \beta^d \qquad \text{Par définition de } (\!|-|\!)^*_{\beta=1} \tag{I.7}$$

$$= (\!|w|\!)^*_{\alpha=1} \times \beta^d \qquad \text{D'après l'inégalité I.4} \tag{I.8}$$

On définit $p$ comme étant le plus grand degré d'un polynôme $(\!|b|\!)^*_1$ et $e$ comme le degré de $(\!|w|\!)^*_\alpha$ en $\alpha$. On va montrer par induction sur la structure d'une expression $w$ que $e$ ,$d$ et $p$ sont liés par l'inégalité suivante $d \leq p \times e + 1$ :

- Si $w$ est un symbole d'arité 0, $(\!|w|\!)^*_\alpha = 1$ et $(\!|w|\!)^*_\beta \leq \beta$. Par conséquent, $d = 1$ et $e = 0$ et l'inégalité est satisfaite.
- Supposons que $w = h(t_1, \cdots, t_n)$ avec $d_j$ le degré de $(\!|t_j|\!)^*_\beta$ et $e_j$ le degré de $(\!|t_j|\!)^*_\alpha$. Par hypothèse d'induction, on a $d_j \leq p \times e_j + 1$. Maintenant supposons que l'écriture de $(\!|h|\!)^*_\beta(X_1, \cdots, X_n)$ se développe en $\sum_{j_1 \leq d_1 \ldots j_n \leq d_n} \beta \times X_1^{j_1} \ldots X_n^{j_n}$. Soient $(i_1, \cdots, i_n)$ les indices dans le polynôme $(\!|h|\!)^*_\beta$ pour lesquels le degré est atteint $d = \sum_{j=1}^{n} d_j i_j + 1$

(ils ne sont pas uniques mais il existe au moins un tel couple).

$$\begin{aligned}
d &\leq \sum_{j=1}^{n}(p \times e_j + 1)i_j + 1 = p\sum_{j=1}^{n} e_j \times i_j + \sum_{j=1}^{n} i_j + 1 && \text{Par H.I.} \\
&\leq p\sum_{j=1}^{n} e_j \times i_j + p + 1 = p \times (\sum_{j=1}^{n} e_j \times i_j + 1) + 1 && \text{Par définition de } p \\
&\leq p \times e + 1 && \text{Par définition de } e
\end{aligned}$$

La combinaison de ce résultat et de l'inégalité I.8 nous donne :

$$(\!|w|\!)^*_\beta \leq (\!|w|\!)^*_{\alpha=1} \times \beta^{p\times e+1} = (\!|w|\!)^*_{\alpha=1} \times \beta(\beta^p)^e \tag{I.9}$$

On choisit un entier $z$ tel que $\alpha^z \geq \beta^p$ et on définit le polynôme $P$ par $P(X) = \beta \times X^{z+1}$. L'entier $z$ existe puisque $\alpha > 1$ et le polynôme $P$ ne dépend pas de l'expression $w$ mais des coefficients des quasi-interprétations.

Il reste alors à vérifier que $(\!|w|\!)^*_1 \leq P((\!|w|\!)^*_2)$ :

$$\begin{aligned}
(\!|w|\!)^*_1 &\leq (\!|w|\!)^*_{\alpha=1} \times \beta(\beta^p)^e && \text{D'après I.9} \\
&\leq (\!|w|\!)^*_{\alpha} \times \beta(\beta^p)^e && \text{Puisque } \alpha > 1 \\
&\leq (\!|w|\!)^*_{\alpha} \times \beta(\alpha^z)^e = (\!|w|\!)^*_{\alpha} \times \beta(\alpha^e)^z && \text{Par définition de } z \\
&\leq (\!|w|\!)^*_{\alpha} \times \beta((\!|w|\!)^{*}_{\alpha})^z) = P((\!|w|\!)^*_{\alpha}) && \text{Puisque } \alpha^e \leq (\!|w|\!)^*_{\alpha} \\
&\leq P((\!|w|\!)^*_2) && \text{D'après I.2}
\end{aligned}$$

□

**Théorème I.82.** *Étant donnés deux programmes $\boldsymbol{p}_1 = \langle Var_1, Cns_1, Fct_1, R_1\rangle$ et $\boldsymbol{p}_2 = \langle Var_2, Cns_2, Fct_2, R_2\rangle$ admettant des quasi-interprétations additivement préservatrices $(\!|-|\!)_1$ et $(\!|-|\!)_2$ et leur union stratifiée $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$, le lemme fondamental est vérifié : Il existe un polynôme $P$ tel que pour toute expression $t$ de $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ possédant $n$ variables $x_1, \cdots, x_n$ et pour toute substitution $\sigma$ vérifiant $x_i\sigma = v_i$, on a :*

$$|[\![t\sigma]\!]| \leq P^{|t|}(\max_{i=1..n} |v_i|)$$

*Démonstration.* La preuve repose sur les deux lemmes précédents. On utilise une induction structurelle sur une expression $t$.

(i) Le cas de base se présente lorsque $t = \mathtt{f}(x_1, \ldots, x_k)$ avec des variables $x_i$. On utilise alors la stratégie d'évaluation décrite dans le lemme I.79. Le calcul s'effectue donc en utilisant au plus $\ell$ alternances : $\mathtt{f}(v_1, \cdots, v_n) = w_0 \overset{*}{\to}_2 u_1 \overset{*}{\to}_1 w_1 \ldots \overset{*}{\to}_2 u_l \overset{*}{\to}_1 w_\ell$, la constante $\ell$ étant bornée par la taille du programme.

Désormais, nous montrons par induction sur l'entier $m \leq l$ qu'il existe un polynôme $S$ tel que $(\!|w_m|\!)_1^* \leq S^m(P((\!|w_0|\!)_2^*))$ (I.H.). Le cas où $m = 0$ est trivial. Définissons $S = R \circ P$ avec $P$ et $R$ les polynômes du lemme I.81. On vérifie alors que :

$$
\begin{aligned}
(\!|w_{m+1}|\!)_2^* &\leq R((\!|w_{m+1}|\!)_1^*) && \text{Lemme I.81} \\
&\leq R((\!|u_{m+1}|\!)_1^*) && \text{Proposition I.19} \\
&\leq R(P((\!|u_{m+1}|\!)_2^*)) && \text{Lemme I.81} \\
&\leq R(P((\!|w_m|\!)_2^*)) && \text{Proposition I.19} \\
&\leq S(S^m(P((\!|w_0|\!)_2^*))) && \text{Par H.I.} \\
&= S^{m+1}(P((\!|w_0|\!)_2^*))
\end{aligned}
$$

En appliquant le lemme fondamental à $\mathbf{p}_2$, on obtient un polynôme $T$ qui vérifie $(\!|w_0|\!)_2^* \leq T(\max_{i=1..n}|v_i|)$. De sorte que :

$$|w_\ell| \leq (\!|w_\ell|\!)_2^* \leq S^\ell(P(T(\max_{i=1..n}|v_i|)))$$

(ii) L'induction découle de l'inégalité ci-dessus par composition des polynômes.

□

**Exemple 12.** *Soit le programme* $\boldsymbol{p}_1$ *défini par :*

$$
\begin{aligned}
\texttt{double}(y) = \textbf{Case}\ y\ \textbf{of}\ &\mathbf{S}(x) \rightarrow \mathbf{S}(\mathbf{S}(\texttt{double}(x))) \\
&\mathbf{0} \rightarrow \mathbf{0} \\
&- - - - - - - - - - - - - \\
\texttt{add}(u, v) = \textbf{Case}\ u, v\ \textbf{of}\ &\mathbf{S}(x), y \rightarrow \mathbf{S}(\texttt{add}(x, y)) \\
&\mathbf{0}, y \rightarrow y
\end{aligned}
$$

*et le programme* $\boldsymbol{p}_2$ *défini ci-dessous :*

$$
\begin{aligned}
\texttt{sq}(y) = \textbf{Case}\ y\ \textbf{of}\ &\mathbf{S}(x) \rightarrow \mathbf{S}(\texttt{add}(\texttt{sq}(x), \texttt{double}(x))) \\
&\mathbf{0} \rightarrow \mathbf{0}
\end{aligned}
$$

*Leur union hiérarchique* $\boldsymbol{p}_1 \ll \boldsymbol{p}_2 = \langle Var, Cns, Fct, \mathcal{R}\rangle$ *calcule le carré d'un nombre unaire donné en entrée. Pour la précédence* $\geq_{Fct}$*, on a* $\texttt{sq} >_{Fct} \{\texttt{add}, \texttt{double}\}$*. De plus, le programme* $\boldsymbol{p}_2$ *est plat puisqu'il n'y a aucune composition de symboles de fonction dans ses définitions. Par conséquent,* $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ *est une union stratifiée, puisque l'argument du seul appel récursif* $\texttt{sq}(x)$ *est une variable.* $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *sont tous les deux ordonnés par* $\prec_{rpo}$

*avec un statut produit. On définit les quasi-interprétations* $(\!|-|\!)_1$ *et* $(\!|-|\!)_2$ *par :*

$$\begin{aligned} (\!|\mathbf{0}|\!)_1 &= 0 \\ (\!|\mathbf{S}|\!)_1(X) &= X+1 \\ (\!|\mathtt{double}|\!)_1(X) &= 2 \times X \\ (\!|\mathtt{add}|\!)_1(X,Y) &= X+Y \end{aligned} \qquad \Bigg| \qquad \begin{aligned} (\!|\mathbf{0}|\!)_2 &= 0 \\ (\!|\mathbf{S}|\!)_2(X) &= X+1 \\ (\!|\mathtt{double}|\!)_2(X) &= 2 \times X \\ (\!|\mathtt{add}|\!)_2(X,Y) &= X+Y+1 \\ (\!|\mathtt{sq}|\!)_2(X) &= 2 \times X^2 \end{aligned}$$

$(\!|-|\!)_1$ *et* $(\!|-|\!)_2$ *sont des quasi-interprétations additivement préservatrices de sorte que le programme* $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ *calcule des valeurs dont la taille est bornée polynomialement par la taille des entrées.*

*De plus, la division du programme peut être itérée sur* $\boldsymbol{p}_1$ *en utilisant le découpage donné en trait pointillé qui sépare les définitions des symboles de fonction* `add` *et* `double`*, engendrant ainsi une union à constructeurs partagés.*

**Corollaire I.83** (TEMPS et ESPACE pour l'union hiérarchique ). *L'ensemble des fonctions calculées par une union hiérarchique de deux programmes* $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *telle que*

1. $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ *est une union stratifiée,*
2. $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *admettent des quasi-interprétations additivement préservatrices* $(\!|-|\!)_1$ *et* $(\!|-|\!)_2$*,*
3. $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *sont ordonnées par* $\prec_{rpo}$ *et chaque symbole de fonction a un statut produit,*

*est exactement l'ensemble des fonctions calculables en temps polynomial.*

*De plus, si la condition (3) est remplacée par :* $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *sont ordonnés par* $\prec_{rpo}$ *alors on caractérise exactement la classe des fonctions calculables en espace polynomial.*

*Démonstration.* Ce résultat est une conséquence du lemme fondamental de la proposition I.82 et de l'ordre $\prec_{rpo}$ avec statut produit. □

Nous avons ainsi établi une nouvelle décomposition des programmes, appelée union stratifiée, qui garantit que le lemme fondamental demeure correct. Une telle décomposition est une condition nécessaire afin de préserver la borne polynomiale, comme nous l'avons illustré dans les différents contre-exemples qui précèdent. Ainsi, on obtient une nouvelle manière de diviser les programmes, encore plus flexibles que les précédentes, dans une perspective de synthèse de quasi-interprétation. La complexité temporelle des heuristiques de recherche de quasi-interprétations se trouve donc considérablement améliorée.

## I.5 Application à d'autres langages de programmation

Dans cette section, nous montrons brièvement que l'analyse des programmes par quasi-interprétation peut être étendue à d'autres types de langage. En particulier, on montrera que les quasi-interprétations ont déjà été adaptées à des problèmes de vérification de bytecode dans [ACGDZJ04]. Puis, nous nous intéressons à une adaptation des quasi-interprétations au contrôle des ressources dans le cadre d'un langage synchrone utilisant des threads coopératifs développée dans [ADZ04]. Enfin, on vérifiera que les résultats obtenus pour la modularité hiérarchique en section I.4 permettent d'élargir notre étude à l'ordre supérieur en utilisant des techniques d'élimination de l'ordre supérieur.

### I.5.1 Application à la vérification de bytecode

Dans [ACGDZJ04], Amadio et al. adaptent l'étude des quasi-interprétations à la vérification des ressources d'un bytecode. À cet effet, ils considèrent un langage fonctionnel au premier ordre standard similaire au langage décrit en section I.1.1 et y ajoutent des types inductifs. Puis, ils définissent une évaluation par appel par valeur similaire à celle présentée en figure I.2. Dans cette étude, on suppose que les quasi-interprétations ainsi que les types sont fournis comme des annotations des programmes. Une notion de machine virtuelle composée de six instructions de base est alors définie. Ces six instructions sont les suivantes :

- `load` $i$, qui charge dans la pile la $i$-ème valeur de la pile,
- `build` $\mathbf{c}$ $n$ qui construit et ajoute la valeur $\mathbf{c}(v_1, \cdots, v_n)$ dans la pile, si les $v_1, \cdots, v_n$ sont les $n$ valeurs situées en haut de la pile au moment où l'instruction est exécutée,
- `branch` $\mathbf{c}$ $j$ qui exécute l'instruction $j$ si la valeur située en haut de la pile est de la forme $\mathbf{c}(v_1, \cdots, v_n)$,
- `stop` qui arrête l'exécution,
- `call` `g` $n$ qui appelle la fonction `g` sur les $n$ valeurs situées en haut de la pile,
- et `return` $n$ qui retourne la valeur calculée par un symbole de fonction `f` d'arité $n$.

Le but de cette étude est de déterminer les vérifications d'usage à effectuer pour garantir des bornes en espace et des preuves de terminaison au niveau du bytecode en utilisant les quasi-interprétations fournies avec le programme initial. Elle s'effectue en trois étapes :

1. On vérifie d'abord que les configurations de la machine virtuelle sont bien typées relativement aux types donnés en annotation,
2. Puis, on effectue une vérification sur leur forme et, en particulier, sur le type de symbole de constructeur contenu en haut de pile,

3. Enfin, On effectue une vérification sur la taille des valeurs à l'aide des annotations de quasi-interprétations.

Si les programmes compilés passent avec succès ces trois étapes d'analyse alors on obtient que la taille de toute valeur stockée dans la pile est bornée polynomialement par la taille des entrées. En outre, l'analyse est étendue à des interprétations polynomiales [MN70, Lan79] afin d'obtenir une vérification de la terminaison de la machine virtuelle. Cette étude a été améliorée dans [DZG05] en utilisant des récurrences terminales.

### I.5.2 Application à des systèmes de threads concurrents et interactifs

Dans [ADZ04], une analyse statique de systèmes de threads interactifs et concurrents utilisant les quasi-interprétations est proposée. Les threads sont coopératifs et leur interaction est basée sur un modèle synchrone. En d'autres termes, un thread en cours d'exécution ne peut pas être préempté à moins qu'il décide lui-même de rendre la main et l'exécution est découpée en instants. Un instant dure aussi longtemps qu'un thread peut continuer à s'exécuter. Lorsque tous les threads sont stoppés ou sont en attente d'un nouvel instant, le programme passe à l'instant suivant. Le langage en considération s'inspire d'un langage fonctionnel au premier ordre avec des types inductifs auquel on ajoute des comportements (Behaviours). Un comportement peut être retourné par une fonction. Cependant, au lieu de retourner un résultat, le comportement peut produire un effet de bord en changeant des valeurs contenues dans des registres globaux et peut changer l'ordre d'exécution dans le programme du thread qui lui correspond. La syntaxe des expressions est identique à celle décrite en section I.1.1 tandis que les différents types de comportement sont décrits ci-dessous :

– `stop` termine l'exécution du thread considéré,
– `yield`.$b$ passe la main au programme. Lorsque le contrôle revient sur ce thread dans le même instant, ce dernier reprend l'exécution du comportement $b$,
– $e$, où $e$ est une expression, change instantanément le comportement du thread,
– `next`.$e$, où $e$ est une expression, change le comportement du thread au début de l'instant suivant,
– $r := e.b$, où $r$ est un registre global et $e$ est une expression, assigne la valeur de $e$ à $r$ puis exécute le comportement $b$,
– `match` $r$ `with` $p_1 \to b_1 \mid \ldots \mid p_l \to b_l \mid [x] \to e$ attend jusqu'à ce que la valeur de $r$ puisse être unifiée avec celle de $p_i$ et rend le contrôle au programme dans la négative. Si à la fin d'un instant, aucune unification n'a eu lieu alors $e\sigma$ est exécuté, avec $\sigma$ une substitution telle que $x\sigma = v$ et avec $v$ la valeur contenue dans le registre $r$.

Un symbole de fonction f de ce langage est décrit par une suite de règles de la forme $\mathtt{f}(\overline{p_1}) = b_1, \ldots, \mathtt{f}(\overline{p_l}) = b_l$, où les $b_k$ sont des comportements. On conserve l'hypothèse que les motifs sont linéaires et ne se chevauchent pas. Une restriction sur les programmes est alors posée en effectuant une analyse de leur flot de contrôle. Cette analyse garantit qu'un registre peut être lu au plus une fois en l'espace d'un instant (Read Once Condition). La notion de quasi-interprétation est alors étendue aux registres globaux puis est vérifiée sur un système de contraintes générées pour tous les comportements ne correspondant pas à un changement d'instant. L'analyse ainsi effectuée garantit l'obtention d'une borne sur la taille des valeurs calculées à la fin d'un instant par la taille des valeurs au début de l'instant. Cette étude a été étendue dans [Dab07] sur des programmes réactifs afin de garantir une borne sur la taille des valeurs calculées polynomiale en la taille du programme initial à la fin d'un instant quelconque.

### I.5.3 Application à des programmes d'ordre supérieur

Dans cette section, nous essayons d'étendre l'analyse de programme par quasi-interprétation à des programmes fonctionnels à l'ordre supérieur. Afin d'effectuer une telle analyse deux approches différentes sont envisageables. D'une part, on pourrait essayer de considérer des assignations à l'ordre supérieur. Cette approche, quoi que prometteuse, semble cependant difficilement automatisable puisqu'elle consiste à synthétiser des assignations à l'ordre supérieur. D'autre part, on peut transformer tout programme à l'ordre supérieur en un programme équivalent au premier ordre à l'aide de techniques de défonctionnalisation introduites par Reynolds [Rey98]. La défonctionnalisation d'un programme se compose de deux étapes. En premier lieu, un symbole de constructeur est substitué à chaque déclaration de symbole de fonction à l'ordre supérieur. Dans un deuxième, chaque application d'un symbole de fonction est éliminé en introduisant un nouveau symbole de fonction pour l'application. Nous invitons le lecteur intéressé par un tel sujet à lire le mémoire d'habilitation de Danvy [Dan06] qui étudie la défonctionnalisation et le Continuation-Passing Style et donne de nombreuses références. Nous constatons avec intérêt que les techniques d'élimination de l'ordre supérieur transforment les programmes à l'ordre supérieur en une union hiérarchique de programmes, puisque l'application d'une fonction est remplacée par l'application d'un symbole de constructeur. Il est donc envisageable d'appliquer les résultats obtenus en section I.4 à de tels programmes obtenant ainsi des caractérisations à l'ordre supérieur des ensembles des fonctions calculables en temps polynomial et des fonctions calculables en espace polynomial.

Nous allons montrer comment utiliser la modularité des quasi-interprétations afin de contrôler les ressources des programmes à l'ordre supérieur en utilisant les résultats

de [BMP07]. Notre objectif est plus de donner un aperçu du type d'analyse applicable sur des programmes à l'ordre supérieur que de répertorier les différentes techniques de défonctionnalisation existantes. Nous allons donner les points-clé de cette étude sans chercher à formaliser la syntaxe des programmes à l'ordre supérieur de manière précise. Nous faisons donc le choix d'illustrer notre approche par de nombreux exemples.

**Exemple 13.** *Supposons que* g *est défini par un programme* $\boldsymbol{q}_3$*. Considérons* $\boldsymbol{p}$*, le programme à l'ordre supérieur suivant :*

$$\begin{aligned}\mathtt{fold}(\lambda x.\mathtt{f}(x), \mathbf{nil}) &\to \mathbf{0}\\ \mathtt{fold}(\lambda x.\mathtt{f}(x), \mathbf{c}(y,l)) &\to \mathtt{f}(\mathtt{fold}(\lambda x.\mathtt{f}(x), l))\\ \mathtt{h}(l) &\to \mathtt{fold}(\lambda x.\mathtt{g}(x), l) \qquad \textit{pour } \mathtt{g} \textit{ défini dans } \boldsymbol{q}_3\end{aligned}$$

$\mathtt{h}(l)$ *itère* g *de sorte que* $[\![\mathtt{h}(l)]\!] = [\![\mathtt{g}^n(\mathbf{0})]\!]$ *où* $n$ *est le nombre d'éléments dans la liste* $l$*. À partir de* $\boldsymbol{p}$*, nous obtenons* $\hat{\mathbf{p}}$ *par défonctionnalisation :*

$$\boldsymbol{q}_1 = \begin{cases}\hat{\mathtt{fold}}(\mathbf{nil}) \to \mathbf{0}\\ \hat{\mathtt{fold}}(\mathbf{c}(y,l)) \to \mathtt{append}(\mathbf{c}_0, \hat{\mathtt{fold}}(l)) & \textit{avec } \mathbf{c}_0 \textit{ un nouveau constructeur}\\ \hat{\mathtt{h}}(l) \to \hat{\mathtt{fold}}(l)\end{cases}$$

$$\boldsymbol{q}_2 = \left\{\mathtt{append}(\mathbf{c}_0, x) \to \mathtt{g}(x)\right.$$

On peut donc assigner des quasi-interprétations aux programmes d'ordre supérieur en considérant leur transformation au premier ordre. En effet, l'exemple ci-dessus illustre le fait que le programme défonctionnalisé $\hat{\mathbf{p}}$ est divisé en trois parties : les programmes $\mathbf{q}_1$ et $\mathbf{q}_2$ et un programme $\mathbf{q}_3$ qui calcule la fonction correspondant au symbole g. Remarquons que l'union hiérarchique $\mathbf{q}_2 \ll \mathbf{q}_1$ est stratifiée (Cf. définition I.77). De plus, elle admet la quasi-interprétation additivement préservatrice suivante :

| | |
|---|---|
| $(\!\|\hat{\mathtt{fold}}\|\!)_1(X) = (\!\|\mathtt{h}\|\!)_1(X) = X$ | $(\!\|\mathtt{g}\|\!)_2(X) = X+1$ |
| $(\!\|\mathbf{c}\|\!)_1(X,Y) = X+Y+1$ | $(\!\|\mathbf{c}_0\|\!)_2 = 0$ |
| $(\!\|\boldsymbol{0}\|\!)_1 = (\!\|\mathbf{c}_0\|\!)_1 = (\!\|\mathbf{nil}\|\!)_1 = 0$ | $(\!\|\mathtt{append}\|\!)_2(X,Y) = X+Y+1$ |
| $(\!\|\mathtt{append}\|\!)_1(X,Y) = X+Y+1$ | |

Les résultats sur la modularité des quasi-interprétations établis en section I.4.3 nous permettent de donner une condition suffisante sur la quasi-interprétation du symbole g définie dans $\mathbf{q}_3$ afin de garantir que les calculs demeurent polynomialement bornés en la taille des entrées. En effet, la proposition I.82 implique que $(\!|\mathtt{g}|\!)_3$ doit être additivement préservatrice. C'est à dire que $(\!|\mathtt{g}|\!)_3(X) = (\!|\mathtt{g}|\!)_2(X) + \alpha = X + \alpha + 1$ pour une constante $\alpha$ donnée. Remarquons que $(\!|\mathtt{g}|\!)_2$ est forcée par $(\!|\mathtt{append}|\!)_2$ et que $(\!|\mathtt{append}|\!)_2$ est forcée par $(\!|\mathtt{append}|\!)_1$.

**Exemple 14.** *Étant donné le programme* $\boldsymbol{p}$ *suivant, qui visite une liste* $l$ *donnée en entrée en utilisant le continuation passing style :*

$$\begin{aligned}\mathtt{visit}(\mathbf{nil}, \lambda x.\mathtt{f}(x), y) &\to \mathtt{f}(y)\\ \mathtt{visit}(\mathbf{c}(z,l), \lambda x.\mathtt{f}(x), y) &\to \mathtt{visit}(l, \lambda x.\mathtt{g}_1(\mathtt{f}(x)), y)\\ \mathtt{h}(l) &\to \mathtt{visit}(l, \lambda x.\mathtt{g}_0(x), 0)\end{aligned}$$

*où* $\mathtt{g}_0$ *et* $\mathtt{g}_1$ *sont des symboles de fonction définis dans un programme* $\boldsymbol{q}_3$ *donné qui admet une quasi-interprétation additive* $(\!|-|\!)_3$. *On a* $[\![\mathtt{h}(l)]\!] = [\![\mathtt{g}_1^n(\mathtt{g}_0(\mathbf{0}))]\!]$ *où* $n$ *est le nombre d'éléments dans la liste* $l$. *On obtient alors* $\hat{\mathbf{p}}$ *par défonctionnalistation :*

$$\boldsymbol{q}_1 = \begin{cases}\hat{\mathtt{visit}}(\mathbf{nil}, k, y) \to \mathtt{append}(k, y)\\ \hat{\mathtt{visit}}(\mathbf{c}(z,l), k, y) \to \hat{\mathtt{visit}}(l, \mathbf{c}_1(k), y)\\ \mathtt{h}(l) \to \hat{\mathtt{visit}}(l, \mathbf{c}_0, 0)\end{cases} \qquad \textit{avec } \mathbf{c}_0,\ \mathbf{c}_1 \textit{ symboles de constructeur}$$

$$\boldsymbol{q}_2 = \begin{cases}\mathtt{append}(\mathbf{c}_0, x) \to \mathtt{g}_0(x)\\ \mathtt{append}(\mathbf{c}_1(k), x) \to \mathtt{g}_1(\mathtt{append}(k, x))\end{cases}$$

*L'union hiérarchique* $\boldsymbol{q}_2 \ll \boldsymbol{q}_1$ *est stratifiée et admet les quasi-interprétations additivement préservatrices suivantes :*

| | |
|---|---|
| $(\!|\hat{\mathtt{visit}}|\!)_1(X) = (\!|\mathtt{h}|\!)_1(X) = X+1$ | $(\!|\mathtt{g}_1|\!)_2(X) = (\!|\mathtt{g}_0|\!)_2(X) = X+1$ |
| $(\!|\mathbf{c}_1|\!)_1(X,Y) = (\!|\mathbf{c}|\!)_1(X,Y) = X+Y+1$ | $(\!|\mathbf{c}_1|\!)_2(X) = X+1$ |
| $(\!|\mathbf{0}|\!)_1 = (\!|\mathbf{c}_0|\!)_1 = (\!|\mathbf{nil}|\!)_1 = 0$ | $(\!|\mathbf{0}|\!)_2 = (\!|\mathbf{c}_0|\!)_2 = 0$ |
| $(\!|\mathtt{append}|\!)_1(X,Y) = X+Y+1$ | $(\!|\mathtt{append}|\!)_2(X,Y) = X+Y+1$ |

*À présent, supposons que nous disposions de deux quasi-interprétations* $(\!|\mathtt{g}_0|\!)_3$ *et* $(\!|\mathtt{g}_1|\!)_3$, *qui sont deux certificats pour les symboles* $\mathtt{g}_0$ *et* $\mathtt{g}_1$ *par rapport au programme* $\boldsymbol{q}_3$. *La proposition I.82 nous assure que le calcul reste polynomial si les quasi-interprétations des symboles* $\mathtt{g}_0$ *et* $\mathtt{g}_1$ *sont additivement préservatrices. En d'autres termes,* $(\!|\mathtt{g}_0|\!)_3(X) = X+\alpha$ *et* $(\!|\mathtt{g}_1|\!)_3(X) = X+\beta$ *avec* $\alpha$ *et* $\beta$ *des constantes données.*

Une approche modulaire permet donc de prédire si on peut appliquer de manière sure et efficace une fonction dans un programme à l'ordre supérieur. Elle nous permet aussi d'obtenir les caractérisations suivantes :

**Théorème I.84** (Modularité des programmes à l'ordre supérieur)**.** *L'ensemble des fonctions calculées par un programme à l'ordre supérieur* $\mathbf{p}$ *tel que sa défonctionnalisation* $\hat{\mathbf{p}}$ *est définie par union hiérarchique* $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ *de deux programmes* $\boldsymbol{p}_1$ *et* $\boldsymbol{p}_2$ *qui satisfont :*

1. $\boldsymbol{p}_1 \ll \boldsymbol{p}_2$ *est une union stratifiée,*

2. *$\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ admettent des quasi-interprétations additivement préservatrices $(\!|-|\!)_1$ et $(\!|-|\!)_2$,*
3. *$\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ sont ordonnés par $\prec_{rpo}$ et chaque symbole de fonction a un statut produit,*

*est exactement l'ensemble des fonctions calculables en temps polynomial.*

*De plus, si la condition (3) est remplacée par : $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ sont ordonnés par $\prec_{rpo}$ alors on caractérise exactement la classe des fonctions calculables en espace polynomial.*

## I.6 Conclusion

La diversité des algorithmes capturés, la décidabilité du problème de la synthèse pour des polynômes à valeur sur les réels et à degrés bornés ainsi que l'application des quasi-interprétations à des langages aussi divers que ceux mentionnés dans la précédente section justifient pleinement l'intérêt que l'on peut porter à une telle méthode d'analyse statique par interprétation sémantique.

Cependant, cette méthode possède quelques lacunes. L'analyse des quasi-interprétations échoue sur certains algorithmes naturels tels que le tri rapide (Quicksort) ou encore la division. Ce phénomène est, en majeure partie, dû à la propriété sous-terme I.13 des assignations considérées dans la définition des quasi-interprétations. De plus, il nous faut fournir une analyse plus flexible permettant à la fois d'analyser les ressources d'un plus grand nombre d'algorithmes, d'améliorer l'étude des programmes réactifs et d'analyser les ressources utilisées par des programmes impératifs ou orientés objet. C'est dans cette optique que nous introduirons la notion de sup-interprétation d'un langage fonctionnel au premier ordre dans le chapitre II, puis, que nous adapterons cet outil à un langage orienté objet dans le chapitre III.

# II Les sup-interprétations

Il n'existe pas qu'un seul monde, lui dis-je, celui que vous voyez ou que vous croyez voir ou que vous imaginez voir ou que vous voulez bien voir, ce monde que touchent les aveugles, qu'entendent les amputés, que reniflent les sourds, ce monde de choses et de forces, de solidités et d'illusions, ce monde de vie et de mort, de naissances et de destructions, ce monde où nous buvons, au milieu duquel nous avons coutume de nous endormir. Il en existe au moins un autre à ma connaissance : celui des nombres et des figures, des identités et des fonctions, des opérations et des groupes, des ensembles et des espaces.

*(R. Queneau, Odile)*

Ce chapitre améliore l'analyse statique de la complexité des programmes du premier ordre effectuée au chapitre I. Il met en évidence la notion de sup-interprétation introduite dans [MP06b, MP07c]. Une sup-interprétation est un outil qui fournit une borne supérieure sur la taille des sorties d'une fonction. Plus précisément, une sup-interprétation est une assignation partielle qui fait correspondre une fonction de domaine et codomaine inclus dans l'ensemble des réels à une partie des symboles d'un programme. La fonction assignée fournit directement une borne sur la taille des valeurs calculées par le symbole correspondant.

Cette notion de sup-interprétation est utilisée dans un critère, dénommé critère quasi-amical, qui garantit que la taille des valeurs calculées par un programme le vérifiant est bornée polynomialement par la taille de ses entrées. L'intérêt pratique d'un tel critère est de fournir, avant exécution, la quantité de ressources, par exemple, l'espace mémoire, qu'un programme nécessitera pendant son exécution.

La notion de sup-interprétation a donc été introduite pour répondre à ce type de besoin. Elle s'inspire fortement de :

- La notion de quasi-interprétation étudiée dans le chapitre précédent et développée par Bonfante, Marion et Moyen dans différents articles [BMM01, BMM07, MM00, Mar03]. Comme nous l'avons vu dans la section I.2, une quasi-interprétation fournit, tout comme une sup-interprétation, une borne supérieure sur la taille des sorties d'une fonction par analyse statique de programmes fonctionnels au premier ordre. Un programmeur connaissant la quasi-interprétation d'un programme peut donc aisément trouver une borne sur la taille de chaque stack frame. Mais l'intérêt principal des sup-interprétations est de capturer une classe strictement plus large d'algorithmes que les quasi-interprétations. Elles ont donc une plus grande complétude intensionnelle. Cette caractéristique est intrinsèquement liée à l'absence de la propriété sous-terme (Cf. définition I.13) pour les sup-interprétations. En effet, cette propriété génère parfois des des bornes supérieures beaucoup trop larges et engendre ainsi une perte de flexibilité dans la recherche de quasi-interprétation. Ainsi, les sup-interprétations fournissent une plus grande flexibilité dans l'étude des algorithmes que les quasi-interprétations. Afin d'illustrer ce propos, nous verrons dans ce chapitre que des programmes naturels calculant le logarithme ou la division admettent une sup-interprétation et n'ont aucune quasi-interprétation.
- La méthode des paires de dépendance développée par Arts and Giesl dans [AG00]. Cette méthode fut initialement introduite pour prouver la terminaison des systèmes de réécriture de termes de manière automatique. Les sup-interprétations s'inspirent fortement des paires de dépendance à travers la notion de fraternité introduite en II.1.2.
- Le size-change principle (ou SCP) de Jones et al. [LJBA01], une autre méthode permettant de prouver la terminaison des programmes. En effet, la relation entre la terminaison d'un programme et sa complexité computationnelle est forte puisque pour prouver la terminaison d'un programme et pour trouver des bornes de complexité sur ses calculs, il faut contrôler les arguments des fonctions apparaissant lors d'un appel récursif.

Après avoir introduit la notion de sup-interprétation, nous développerons trois critères pour borner les ressources que nous décrivons brièvement :

1. Le premier critère, appelé critère quasi-amical, permet d'agrandir en pratique la classe des programmes capturés par un ancien critère, nommément le critère amical, introduit dans [MP06b]. Par exemple, il permet de capturer des programmes récursifs sur des structures d'arbre alors que le critère amical échoue sur ce type d'algorithme. De plus, il permet de capturer strictement plus d'algorithmes que les quasi-interprétations puisque toute quasi-interprétation est quasi-amicale. L'ob-

jectif de ce critère est de fournir une borne polynomiale sur la taille des valeurs calculées pendant l'exécution d'un programme.

2. Le deuxième critère est appelé critère quasi-amical à appels récursifs bornés. Il permet de prendre en compte des programmes ne terminant pas. Ce second critère fournit, lui aussi, une borne polynomiale sur la taille des valeurs calculées ainsi que sur la taille des stack frames empilés durant l'exécution du programme correspondant. En particulier, il permet l'étude de programmes sur des structures de données infinies, comme les streams, en vérifiant que chaque étape du calcul est bornée polynomialement.

3. Finalement, le dernier critère quasi-amical modulo projection a été introduit pour étudier les programmes utilisant des opérations de destruction ou de projection. En particulier, nous capturons, grâce à ce nouveau critère, de nombreux algorithmes basés sur des techniques de programmation de type « diviser pour régner », algorithmes dont la complexité n'était capturée ni par le critère quasi-amical, ni par aucune autre méthode à notre connaissance.

Puis, nous comparons la notion de sup-interprétation avec :

- La notion de quasi-interprétation. Premièrement, nous montrerons que toute quasi-interprétation est une sup-interprétation particulière. Il en découle que nous obtenons des heuristiques permettant de synthétiser des sup-interprétations. Ces heuristiques permettent de trouver la sup-interprétation d'un programme en cherchant une quasi-interprétation du programme. Une autre conséquence est que toutes les caractérisations de classes de complexité obtenues en section I.2 peuvent être étendues aux sup-interprétations. Enfin, il est donc envisageable d'appliquer les sup-interprétations, tout comme les quasi-interprétations, à d'autres langages bytecode [ACGDZJ04], à l'ordre supérieur ou synchrones avec des threads coopératifs [ADZ04].
- La méthode des paires de dépendance. Nous développons un critère de terminaison grâce à cette notion de paire de dépendance. Ce critère de terminaison utilise des polynômes sur les entiers naturels de manière à préserver le fait que le préordre utilisé par les paires de dépendance soit bien-fondé. Combiné au critère quasi-amical, il permet de caractériser de manière différente l'ensemble des fonctions calculables en espace polynomial.
- Le size-change principle (SCP) afin d'obtenir un autre critère de terminaison. Les programmes dont la terminaison est capturée par le SCP sont aussi capturés par cette méthode mais la réciproque est fausse.

Nous présenterons une caractérisation des classes de complexité $NC^k$ qui représentent les

classes de fonctions calculables par des circuits de profondeur logarithmique et de taille polynomiale. La caractérisation de la classe *ALogTime* a été présentée dans [BMP06] et nous généralisons ce résultat aux classes $NC^k$. De telles caractérisations dépendent fortement de l'utilisation des sup-interprétations, puisque l'absence de condition sous-terme sur les sup-interprétations nous permet facilement de borner logarithmiquement les appels récursifs d'un programme tandis que les quasi-interprétations ne permettent pas une telle approche.

Enfin, nous terminerons ce chapitre en proposant un critère utilisant des assignations polynomiales sans la propriété sous-terme et s'inspirant de la méthode des paires de dépendance qui permet de synthétiser des sup-interprétations n'ayant pas la propriété sous-terme.

## II.1 Définitions

### II.1.1 Adjonction d'opérateurs

Nous enrichissons ici la syntaxe du langage de programmation fonctionnel au premier ordre décrit dans la section I.1.1 à l'aide de symboles d'opérateur. Un opérateur correspondra à une opération primitive incluse par défaut dans le langage étudié. Le vocabulaire $\Sigma = \langle \mathit{Var}, \mathit{Cns}, \mathit{Op}, \mathit{Fct} \rangle$ se compose désormais de quatre ensembles disjoints de symboles qui représentent respectivement l'ensemble des variables, l'ensemble des symboles de constructeur, l'ensemble des symboles d'opérateur et l'ensemble des symboles de fonction. Un programme $\mathbf{p}$ de notre langage se compose d'une séquence de définitions $\mathit{def}_1, \cdots, \mathit{def}_m$ qui décrivent des symboles de fonction et qui sont caractérisées par la grammaire de la figure I.1 où les expressions sont désormais définies par :

$$\begin{aligned} \texttt{Expression} \ni e \quad ::= \quad & x \mid \mathbf{c}(e_1, \cdots, e_n) \mid \mathbf{op}(e_1, \cdots, e_n) \mid \mathtt{f}(e_1, \cdots, e_n) \\ & \mid \mathbf{Case}\ x_1, \cdots, x_n\ \mathbf{of}\ \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell \end{aligned}$$

où $x, x_1, \ldots, x_n$ sont des variables, $\mathbf{c} \in \mathit{Cns}$ est un symbole de constructeur, $\mathbf{op} \in \mathit{Op}$ est un symbole d'opérateur, $\mathtt{f} \in \mathit{Fct}$ est un symbole de fonction et $\overline{p_i}$ est une séquence de $n$ motifs.

FIG. II.1: Syntaxe des expressions avec opérateurs

Les restrictions sur le langage sont les mêmes qu'en section I.1.1. Le domaine de calcul d'un programme $\mathbf{p}$ est toujours l'ensemble $\mathcal{V}^* = \mathcal{V} \cup \{\mathbf{Err}\}$ où $\mathcal{V}$ représente l'algèbre

de termes constructeurs $\mathcal{T}(Cns)$, défini par $\mathcal{T}(Cns) \ni v ::= b \mid b(v_1, \cdots, v_n)$, et **Err** est un symbole sémantique d'arité nulle retourné comme sortie par un programme si une erreur advient. Chaque symbole d'opérateur **op** d'arité $n$ est interprété par une fonction $[\![\mathbf{op}]\!]$ de $\mathcal{V}^n$ dans $\mathcal{V}^*$. Les opérateurs sont essentiellement des fonctions partielles, comme les destructeurs, ou des fonctions caractéristiques de prédicats, comme $=$. Le symbole d'opérateur **hd**, qui calcule la tête d'une liste, illustre l'utilité du symbole **Err** en satisfaisant la condition suivante $[\![\mathbf{hd}]\!](\mathbf{nil}) = \mathbf{Err}$ lorsque **nil** est la liste vide.

Notre langage possède la même sémantique d'appel par valeur définie dans la figure I.2 enrichie de la règle suivante :

$$\frac{t_1 \downarrow w_1 \ldots t_n \downarrow w_n}{\mathbf{op}(t_1, \cdots, t_n) \downarrow [\![\mathbf{op}]\!](w_1, \cdots, w_n)} \; \mathbf{op} \in Op \text{ et } \forall i, w_i \neq \mathbf{Err}$$

FIG. II.2: Sémantique par appel par valeur d'un opérateur

La sémantique d'une expression reste inchangée, i.e. étant donnée une expression sans variable, on note $[\![t]\!] = u$ si et seulement si $t \downarrow u$.

### II.1.2 Fraternités

Dans cette section, nous définissons la notion syntaxique de fraternité inspirée par les deux techniques de terminaison que sont les paires de dépendance [AG00] et le Size-change principle de [LJBA01]. La fraternité est une notion clé utilisée dans différents critères afin de contrôler la taille des arguments des appels récursifs.

**Définition II.1** (Fraternité)**.** *Dans un programme* $\boldsymbol{p}$*, l'expression* $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ *activée par* $\mathtt{f}(p_1, \cdots, p_n)$ *est une fraternité si :*

1. $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ *est une expression maximale (Cf. définition I.4).*
2. *Pour tout* $i \in \{1, \ldots, r\}$*,* $\mathtt{g}_i \approx_{Fct} \mathtt{f}$*.*
3. *Pour tout symbole de fonction* $\mathtt{h}$ *apparaissant dans le contexte* $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$*, on a* $\mathtt{f} >_{Fct} \mathtt{h}$*.*

Une fraternité correspond à un ou plusieurs appels récursifs puisqu'elle fait intervenir des symboles de fonction équivalents pour la précédence $\geq_{Fct}$.

**Exemple 15.** *Le programme de l'exemple 1 :*

$$
\begin{aligned}
\mathtt{minus}(x,y) &= \textbf{Case } x,y \textbf{ of}\\
&\quad \mathbf{0}, z \to \mathbf{0}\\
&\quad \mathbf{S}(z), \mathbf{0} \to \mathbf{S}(z)\\
&\quad \mathbf{S}(u), \mathbf{S}(v) \to \mathtt{minus}(u,v)\\
\mathtt{q}(x,y) &= \textbf{Case } x,y \textbf{ of}\\
&\quad \mathbf{0}, \mathbf{S}(z) \to \mathbf{0}\\
&\quad \mathbf{S}(z), \mathbf{S}(u) \to \mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u), \mathbf{S}(u)))
\end{aligned}
$$

*admet deux fraternités qui sont* $\mathtt{minus}(u,v)$ *et* $\mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u),\mathbf{S}(u)))$*. Elles sont respectivement activées par* $\mathtt{minus}(\mathbf{S}(u),\mathbf{S}(v))$ *et* $\mathtt{q}(\mathbf{S}(z),\mathbf{S}(u))$*.*

**Définition II.2.** *Une fraternité* $\mathsf{C}[\mathtt{g}_1(\overline{e_1}),\ldots,\mathtt{g}_r(\overline{e_r})]$ *activée par* $\mathtt{f}(p_1,\cdots,p_n)$ *est dans sa forme canonique si tout symbole de fonction* $\mathtt{h}$ *apparaissant dans* $\mathsf{C}[\diamond_1,\cdots,\diamond_r]$*, on a* $\mathtt{f} >_{Fct} \mathtt{h}$*. Dans la suite de ce chapitre, chaque fois que nous aurons recours à une fraternité* $\mathsf{C}[\mathtt{g}_1(\overline{e_1}),\ldots,\mathtt{g}_r(\overline{e_r})]$*, nous supposerons au préalable qu'elle est dans sa forme canonique.*

### II.1.3 Assignations partielles

**Définition II.3.** *Une assignation* $I$ *est une application partielle de domaine dom*$(I)$ *inclus dans le vocabulaire* $\Sigma$ *qui associe une fonction partielle* $I(b)$ *de* $(\mathbb{R}^+)^n$ *dans* $\mathbb{R}^+$ *à chaque symbole* $b$ *d'arité* $n$ *appartenant au domaine dom*$(I)$*. Afin de simplifier le présent discours, nous supposerons désormais qu'une assignation partielle est toujours définie pour les symboles de constructeur et d'opérateur (i.e.* $Cns \cup Op \subseteq dom(I)$*).*

*Une assignation* $I$ *est définie sur une expression* $e$ *si chaque symbole de* $Cns \cup Op \cup Fct$ *apparaissant dans l'expression* $e$ *appartient au domaine dom*$(I)$*. Si l'assignation* $I$ *est définie sur une expression* $e$ *ayant* $k$ *variables alors l'assignation partielle de* $e$ *par rapport à* $I$*, que nous notons* $I^*(e)$*, est l'extension canonique de l'assignation* $I$*. Elle dénote une fonction de* $(\mathbb{R}^+)^k$ *dans* $\mathbb{R}^+$ *et est définie comme suit :*

1. *Si* $x_i$ *appartient à Var, alors* $I^*(x_i) = X_i$ *avec* $X_1,\ldots,X_k$ *une séquence de nouvelles variables à valeur dans* $\mathbb{R}^+$*.*
2. *Si* $b$ *est un symbole d'arité nulle, alors* $I^*(b) = I(b)$*.*
3. *Si* $b$ *est un symbole d'arité* $n > 0$ *et si* $e_1,\cdots,e_n$ *sont des expressions, alors* $I^*(b(e_1,\cdots,e_n)) = I(b)(I^*(e_1),\ldots,I^*(e_n))$

*Si* $\overline{e}$ *est une séquence d'expressions* $e_1, \cdots, e_m$ *alors* $I^*(\overline{e}) = I^*(e_1), \ldots, I^*(e_m)$.

*Étant donné un contexte* $\mathsf{C}[\diamond_1, \cdots, \diamond_l]$, $I^*(\mathsf{C})$ *est défini comme la fonction de* $(\mathbb{R}^+)^l \to \mathbb{R}^+$ *qui satisfait, pour toute séquence d'expression* $e_1, \cdots, e_l$, $I^*(\mathsf{C})(\theta^*(e_1), \ldots, \theta^*(e_l)) = I^*(\mathsf{C}[e_1, \cdots, e_l])$.

Les notions d'assignations polynomiale et additive restent identiques à celles du chapitre I. Par conséquent, le lemme suivant est toujours valide :

**Lemme II.4.** *Étant donnée une assignation additive* $I$*, il existe une constante* $\alpha$ *telle que pour toute valeur* $v$ *de* $\mathcal{V}^*$*, les inégalités suivantes sont satisfaites :*

$$|v| \leq I^*(v) \leq \alpha \times |v|$$

*Démonstration.* Cf. la preuve du lemme I.12. □

### II.1.4 Sup-interprétations

**Définition II.5.** *Une sup-interprétation est une assignation* $\theta$ *(Cf. définition II.3) qui vérifie les trois conditions suivantes :*

1. *L'assignation* $\theta$ *est monotone. C'est à dire, pour tout symbole* $b \in dom(\theta)$*, la fonction* $\theta(b)$ *satisfait :*

$$\forall i = 1, \ldots, n \; X_i \geq Y_i \Rightarrow \theta(b)(X_1, \cdots, X_n) \geq \theta(b)(Y_1, \cdots, Y_n)$$

2. *Pour toute valeur* $v$*, la sup-interprétation de* $v$ *est supérieure ou égale à la taille de* $v$ *:*

$$\forall v \in \mathcal{V}, \; \theta^*(v) \geq |v|$$

3. *Pour tout symbole* $b \in dom(\theta)$ *d'arité* $n$ *et pour toutes valeurs* $v_1, \ldots, v_n$ *de* $\mathcal{V}$*, si* $[\![b(v_1, \ldots, v_n)]\!] \in \mathcal{V}^*$*, alors*

$$\theta^*(b(v_1, \ldots, v_n)) \geq \theta^*([\![b(v_1, \ldots, v_n)]\!])$$

*Une expression* $e$ *admet une sup-interprétation* $\theta$ *si* $\theta$ *est une assignation définie sur* $e$*. La sup-interprétation d'une expression* $e$ *par rapport à* $\theta$ *est notée* $\theta^*(e)$*.*

Intuitivement, une sup-interprétation est une interprétation particulière d'un programme. Au lieu de retourner la dénotation du programme, la sup-interprétation fournit une borne supérieure sur la taille des sorties des fonctions dénotées par le programme. Il est intéressant de constater qu'une sup-interprétation est une mesure de complexité

à la Blum [Blu67]. En effet, la restriction d'une sup-interprétation[1] aux entiers naturels vérifie les deux axiomes définissant une mesure de complexité. À savoir :

- Étant donné un programme **p** de symbole de fonction principale $\mathtt{f}$, $\theta(\mathtt{f})$ est définie sur les entrées $\theta^*(\overline{v})$ si et seulement si $\mathtt{f}(\overline{v})$ termine.
- Étant donnés le nombre $n$, les entrées $\overline{v}$ et un programme **p** de symbole de fonction principale $\mathtt{f}$, le problème de décision $\phi(\mathtt{f}, \overline{v}, n)$ qui retourne 1 si $\theta(\mathtt{f})(\theta^*(\overline{v})) = n$ et 0 dans le cas contraire est décidable.

**Exemple 16.** *Le programme de l'exemple 1 admet une sup-interprétation polynomiale définie par :*

$$\begin{aligned} \theta(\mathbf{0}) &= 0 \\ \theta(\mathbf{S})(X) &= X + 1 \\ \theta(\mathtt{minus})(X, Y) &= X \\ \theta(\mathtt{q})(X, Y) &= X \end{aligned}$$

**Exemple 17.** *Considérons le programme qui calcule l'exponentielle :*

$$\begin{aligned} \mathtt{exp}(x) \; &= \mathbf{Case}\ x\ \mathbf{of} \\ &\mathbf{0} \to \mathbf{S}(\mathbf{0}) \\ &\mathbf{S}(y) \to \mathtt{double}(\mathtt{exp}(y)) \\ \mathtt{double}(x) \; &= \mathbf{Case}\ x\ \mathbf{of} \\ &\mathbf{0} \to \mathbf{0} \\ &\mathbf{S}(y) \to \mathbf{S}(\mathbf{S}(\mathtt{double}(y))) \end{aligned}$$

*En prenant $\theta(\mathbf{0}) = 0$, $\theta(\mathbf{S})(X) = X + 1$, $\theta(\mathtt{double})(X) = 2X$ et $\theta(\mathtt{exp})(X) = 2^X$, on définit une sup-interprétation des symboles de fonction* double *et* exp*. Notons que la sup-interprétation de* exp *n'est pas polynomiale. Les trois conditions de la définition II.5 se vérifient aisément. Par exemple, en notant $\underline{n} = \underbrace{\mathbf{S}(\ldots\mathbf{S}(\mathbf{0})\ldots)}_{n \text{ fois } \mathbf{S}}$, on vérifiera que $\theta^*(\underline{n}) = n$ et que $\theta^*(\mathtt{double}(\underline{n})) \geq \theta^*([\![\mathtt{double}]\!](\underline{n}))$ puisque :*

$$\begin{aligned} \theta^*(\mathtt{double}(\underline{n})) &= \theta(\mathtt{double})(\theta^*(\underline{n})) = 2 \times \theta^*(\underline{n}) = 2 \times n \\ \theta^*([\![\mathtt{double}]\!](\underline{n})) &= \theta^*(\underline{2 \times n}) = 2 \times n \end{aligned}$$

**Lemme II.6.** *Étant donnée une expression $e$ sans variable admettant une sup-interprétation $\theta$, si $[\![e]\!] \in \mathcal{V}^*$, alors l'inégalité suivante est vérifiée :*

$$\theta^*([\![e]\!]) \leq \theta^*(e)$$

[1] où $\theta(\mathtt{f})(\theta^*(\overline{v}))$ n'est pas défini si $\mathtt{f}(\overline{v})$ ne termine pas.

*Démonstration.* La preuve repose sur une induction structurelle sur les expressions. Le cas de base est une conséquence directe de la propriété suivante : $\forall v \in \mathcal{V},\ [\![v]\!] = v$.

Étant donnée une expression $e = b(e_1, \cdots, e_n)$ admettant une sup-interprétation $\theta$. Nous évaluons $e$ à l'aide d'un appel par valeur. Nous pouvons faire usage d'une telle évaluation puisque les programmes considérés sont confluents. Supposons, par hypothèse d'induction (HI), que $\theta^*(e_i) \geq \theta^*([\![e_i]\!])$. On vérifie que :

$$\begin{aligned}
\theta^*(e) &= \theta(b)(\theta^*(e_1), ..., \theta^*(e_n)) && \text{Par définition de } \theta^* \\
&\geq \theta(b)(\theta^*([\![e_1]\!]), ..., \theta^*([\![e_n]\!])) && \text{Définition II.5 de la condition 1 et (HI)} \\
&= \theta^*(b([\![e_1]\!], ..., [\![e_n]\!])) && \text{Par définition de } \theta^* \\
&\geq \theta^*([\![b]\!]([\![e_1]\!], ..., [\![e_n]\!])) && \text{Définition II.5 de la condition 3} \\
&= \theta^*([\![e]\!])
\end{aligned}$$

□

Étant donnée une expression $e$, on définit $\|e\|$ par :

$$\|e\| = \begin{cases} |[\![e]\!]| & \text{si } [\![e]\!] \in \mathcal{V}^* \\ 0 & \text{sinon} \end{cases}$$

De sorte qu'il est désormais possible de considérer des programmes non terminants de manière élégante :

**Corollaire II.7.** *Supposons que $e$ soit une expression sans variable et admettant une sup-interprétation $\theta$. Alors l'inégalité suivante est vérifiée :*

$$\|e\| \leq \theta^*(e)$$

*Démonstration.* Le cas où $[\![e]\!] \notin \mathcal{V}^*$ est trivial. A présent, supposons que $[\![e]\!] \in \mathcal{V}^*$.

$$\begin{aligned}
\theta^*(e) &\geq \theta^*([\![e]\!]) && \text{Lemme II.6} \\
&\geq \|e\| && \text{Condition 2 de la définition II.5}
\end{aligned}$$

□

### II.1.5 Poids

Désormais, définissons la notion de poids permettant le contrôle de la taille des arguments d'un appel récursif. Le poids est une assignation partielle ayant la propriété sous-terme. La notion de poids est donc très proche de la notion de quasi-interprétation.

**Définition II.8.** *Un poids $\omega$ est une assignation partielle à valeur sur Fct. Étant donné un symbole de fonction $\mathtt{f}$ d'arité $n$, le poids lui associe une fonction totale $\omega_{\mathtt{f}}$ de $(\mathbb{R}^+)^n$ dans $\mathbb{R}^+$ qui vérifie :*

1. *$\omega_{\mathtt{f}}$ est monotone :*

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

2. *$\omega_{\mathtt{f}}$ possède la propriété sous-terme :*

$$\forall i = 1, \ldots, n,\ \forall X_i \in \mathbb{R}^+\ \omega_{\mathtt{f}}(\ldots, X_i, \ldots) \geq X_i$$

## II.2 Critères permettant le contrôle des ressources en espace

Comme nous l'avons mentionné dans l'introduction de cette section, nous allons introduire différents critères combinant des sup-interprétations et des poids polynomiaux. Ces critères permettent de borner la taille des valeurs calculées par un programme polynomialement en la taille des entrées. Le critère principal, appelé quasi-amical, est inspiré par un ancien critère, le critère amical, introduit dans [MP06b]. Cependant, le critère quasi-amical capture plus d'algorithmes. Par exemple, une récurrence sur une structure d'arbre de la forme $\mathtt{f}(x) = \textbf{Case}\ x\ \textbf{of}\ t{*}t' \to \mathtt{f}(t){*}\mathtt{f}(t')$ est quasi-amicale mais n'est pas capturée par le critère amical. Nous cherchons ici à garantir une forte intensionnalité tout en simplifiant au maximum notre discours afin d'éviter toute technicité superflue. C'est pour cette raison que nous n'étudierons pas cet ancien critère. Enfin, les deux derniers critères présentés dans cette section s'inspirent du critère quasi-amical et ont été développés afin de capturer deux catégories d'algorithmes. La première catégorie concerne les algorithmes non-terminants dont, en particulier, les algorithmes sur les structures de données infinies. Nous montrerons que l'on peut tout de même garantir des bornes polynomiales sur la taille de chaque stack frame. La seconde catégorie est constituée d'algorithmes basés sur des techniques de programmation dites « diviser pour régner ». Par exemple, on réussit à capturer l'algorithme du tri rapide (Quick-sort) à l'aide d'un tel critère.

### II.2.1 Critère quasi-amical

**Définition II.9.** *Un programme* $\boldsymbol{p}$ *est* quasi-amical *si et seulement s'il existe une sup-interprétation polynomiale et additive $\theta$ et un poids polynomial $\omega$ tels que pour toute fraternité de la forme $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$, activée par $\mathtt{f}(p_1, \cdots, p_n)$, on ait :*

$$\omega_{\mathtt{f}}(\theta^*(p_1), \ldots, \theta^*(p_n)) \geq \theta^*(\mathsf{C})(\omega_{\mathtt{g}_1}(\theta^*(\overline{e_1})), \ldots, \omega_{\mathtt{g}_r}(\theta^*(\overline{e_r})))$$

**Remarque 3.** *Ce critère n'est pas pertinent lorsque l'on considère la composition de fonctions équivalentes pour la précédence $\geq_{Fct}$, comme l'illustre la définition suivante* $\mathtt{f}(x) = \mathtt{f}(\mathtt{f}(x))$. *Afin de vérifier le critère quasi-amical, il nous faut trouver un poids polynomial $\omega_{\mathtt{f}}$ et une sup-interprétation polynomiale $\theta(\mathtt{f})$ vérifiant :*

$$\omega_{\mathtt{f}}(X) \geq \omega_{\mathtt{f}}(\theta(\mathtt{f})(X))$$

*Ce qui signifie que nous connaissons déjà une borne sur les calculs effectués par le symbole de fonction $\mathtt{f}$, dans la mesure où nous connaissons sa sup-interprétation. Le critère perd donc tout son intérêt dans une telle configuration. Cette restriction ne représente pas un inconvénient majeur pour deux raisons principales. La première étant que de tels programmes ne sont pas naturels du point de vue d'un programmeur. La seconde veut qu'en pratique des programmes de ce type calculent des fonctions plus complexes que les polynômes, comme la fonction d'Ackermann dans l'exemple 18, et tombent, par conséquent, en dehors du champ de notre étude.*

**Exemple 18** (Fonction d'Ackermann [Ack28])**.**

$$\begin{aligned}\mathtt{A}(x, y) = \mathbf{Case}\ x, y\ \mathbf{of}\ \mathbf{0}, z &\to \mathbf{S}(z)\\ \mathbf{S}(z), \mathbf{0} &\to \mathtt{A}(z, \mathbf{S}(\mathbf{0}))\\ \mathbf{S}(z), \mathbf{S}(w) &\to \mathtt{A}(z, \mathtt{A}(\mathbf{S}(z), w))\end{aligned}$$

*Pour vérifier le critère quasi-amical, on doit trouver un poids et une sup-interprétation polynomiale vérifiant $\omega_{\mathtt{A}}(\theta^*(\mathbf{S}(z)), \theta^*(\mathbf{S}(w))) \geq \omega_{\mathtt{A}}(\theta^*(z), \theta^*(\mathtt{A}(\mathbf{S}(z)), \theta^*(w)))$. Ce qui signifie que nous devons trouver $\theta(\mathtt{A})$ afin de vérifier le critère quasi-amical et de trouver une borne sur les calculs effectués par le programme. Le critère perd donc tout son intérêt.*

**Théorème II.10.** *Supposons que le programme $\mathbf{p}$ soit quasi-amical, alors pour tout symbole de fonction $\mathtt{f}$ d'arité $n$ dans $\mathbf{p}$, il existe un polynôme $P_{\mathtt{f}}$ tel que pour toute séquence de valeurs $v_1, \ldots, v_n$, on ait :*

$$\|\mathtt{f}(v_1, \ldots, v_n)\| \leq P_{\mathtt{f}}(\max(|v_1|, ..., |v_n|))$$

*Démonstration.* Supposons que le programme $\mathbf{p}$ soit quasi-amical et que $[\![\mathtt{f}]\!](v_1, \cdots, v_n)$ soit défini, i.e. le calcul termine sur les entrées $v_1, \cdots, v_n$.

Commençons par assigner à chaque motif $p$ une fonction $P_p(X)$ de **Max-Poly** à une variable $X$ :

- si $p$ est une variable alors $P_p(X) = X$
- si $p = \mathbf{c}(p_1, \cdots, p_n)$ alors $P_p(X) = n \times \max_{i=1..n}(P'_{p_i}(X)) + 1$

Par construction, nous obtenons pour toute substitution $\sigma$ satisfaisant $x_i\sigma = v_i$, avec $x_i$ une variable apparaissant dans le motif $p$ :

$$P_p(\max(|v_1|, \cdots, |v_n|)) \geq |p\sigma| = \|p\sigma\| \tag{II.1}$$

À présent, nous allons démontrer ce théorème par induction sur la précédence $\geq_{Fct}$.

- Si le symbole de fonction $\mathtt{f}$ est défini sans fraternité, alors on a $\mathtt{f}(x_1, \cdots, x_n) =$ **Case** $x_1, \cdots, x_n$ **of** $\overline{p^1} \to e_1 \dots \overline{p^l} \to e_l$ avec $\mathtt{f} >_{Fct} \mathtt{g}$ pour tous les symboles de fonction $\mathtt{g} \in e_j$, $j \in \{1, \dots, l\}$. Supposons par hypothèse d'induction, que nous ayons déjà défini une borne supérieure polynomiale sur le calcul des symboles de fonction $\mathtt{g}$. Si $e_j = \mathtt{h}(d_1, \cdots, d_m)$, nous définissons inductivement une borne supérieure sur la taille des calculs de $e_j$ par $P_{e_j}(X) = P_{\mathtt{h}}(\max_{i=1..m} P_{d_i}(X))$, puis, nous posons $P_{\mathtt{f}}(X) = \max_{j=1..l}(P_{e_j}(X))$. Par construction, nous obtenons que $P_{\mathtt{f}}(\max_{i=1..n} |v_i|) \geq \|\mathtt{f}(v_1, \cdots, v_n)\|$ en combinant l'hypothèse d'induction et le cas de base contenu dans l'inéquation (II.1).
- À présent, supposons que le symbole de fonction $\mathtt{f}$ soit défini par des fraternités. Soit $E$ l'ensemble de toutes les expressions maximales activées par $\mathtt{f}(p_1, \cdots, p_n)$ pour une séquence de motifs $p_1, \cdots, p_n$ quelconque et qui ne sont pas des fraternités. Pour toute expression $e \in E$, on commence par définir le polynôme $P_e$, comme dans le paragraphe précédent, puis, on définit le polynôme $P_{\mathtt{f}>}$ par $P_{\mathtt{f}>}(X) = \max_{e \in E}(P_e(X))$. Pour tout symbole de fonction $\mathtt{g} \approx_{Fct} \mathtt{f}$, nous définissons $P_{\mathtt{g}>}$ de manière identique. Enfin, on définit un nouveau polynôme monotone :

  $$Q_{\mathtt{f}}(X) = \max_{\mathtt{g} \approx_{Fct} \mathtt{f}}(P_{\mathtt{g}>}(X))$$

  Intuitivement, ce dernier polynôme représente une borne supérieure sur la taille de toute valeur calculée lors d'un appel de fonction quittant une récurrence, c'est-à-dire, une définition de symbole de fonction qui appelle d'autres symboles de fonction strictement plus petits pour la précédence $\geq_{Fct}$.

  En combinant les inégalités du critère quasi-amical, nous allons établir que si, étant données des valeurs $v_1, \cdots, v_n$, $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{*} \mathsf{C}^*[\mathtt{g}_1(\overline{u_1}), \dots, \mathtt{g}_r(\overline{u_r})]$ avec $\mathtt{g}_1 \approx_{Fct} \dots \approx_{Fct} \mathtt{g}_r \approx_{Fct} \mathtt{f}$, $\mathsf{C}^*$ un contexte correspondant à la composition de différents contextes de fraternité et $\xrightarrow{*}$ la relation de réduction induite par les définitions du programme, alors :

  $$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \theta^*(\mathsf{C}^*)(\omega_{\mathtt{g}_1}(\theta^*(\overline{u_1})), \dots, \omega_{\mathtt{g}_r}(\theta^*(\overline{u_r}))) \tag{II.2}$$

  Ce résultat est établi par induction sur le nombre $k$ d'étapes de réduction correspondant à l'évaluation des symboles de fonction équivalents à $\mathtt{f}$. Si $k = 1$, alors

on obtient le résultat en combinant le critère quasi-amical avec le lemme II.5 et le fait que les poids soient monotones. À présent, supposons que le résultat soit vérifié pour $k > 1$, c'est-à-dire pour une réduction de la forme $\mathtt{f}(\overline{v}) \xrightarrow{k} \mathsf{E}[\mathtt{g}_1(\overline{e_1}), \dots, \mathtt{g}_r(\overline{e_r})]$. De plus, supposons que $\overline{[\![e_j]\!]} = \overline{u_j}$, pour tout $j \in \{1, \dots, r\}$. On obtient alors que $\mathtt{f}(\overline{v}) \xrightarrow{k} \mathsf{E}[\mathtt{g}_1(\overline{u_1}), \dots, \mathtt{g}_r(\overline{u_r})]$ puisque l'évaluation des expressions $e_j$ n'implique aucun symbole de fonction équivalent à $\mathtt{f}$ pour la précédence $\geq_{Fct}$ (cf. Remarque 18). Supposons, respectivement à notre stratégie d'évaluation, que la réduction suivante soit de la forme $\mathtt{g}_j(\overline{u_j}) \xrightarrow{1} \mathsf{D}[\mathtt{h}_1(d_1), \dots, \mathtt{h}_m(d_m)]$ avec $\mathsf{D}[\mathtt{h}_1(d_1), \dots, \mathtt{h}_m(d_m)]$ une fraternité et avec $\mathtt{h}_i \approx_{Fct} \mathtt{g}_j$ pour tout $i \in \{1, \dots, m\}$, de telle sorte que nous puissions appliquer le critère quasi-amical. Par monotonie des sup-interprétations, nous obtenons :

$$\begin{aligned}
\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) &\geq \theta^*(\mathsf{E})(\omega_{\mathtt{g}_1}(\theta^*(\overline{e_1})), \dots, \omega_{\mathtt{g}_r}(\theta^*(\overline{e_r}))) && \text{Par H.I.} \\
&\geq \theta^*(\mathsf{E})(\omega_{\mathtt{g}_1}(\theta^*(\overline{u_1})), \dots, \omega_{\mathtt{g}_r}(\theta^*(\overline{u_r}))) && \text{Lemme II.6} \\
&\geq \theta^*(\mathsf{C}^*)(\omega_{\mathtt{f}_1}(\theta^*(\overline{b_1})), \dots, \omega_{\mathtt{f}_s}(\theta^*(\overline{b_s}))) && \text{Définition II.9}
\end{aligned}$$

pour peu que $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{k+1} \mathsf{C}^*[\mathtt{f}_1(b_1), \dots, \mathtt{f}_s(b_s)]$ avec $s = r + m - 1$, $\mathsf{C}^*[\diamond_1, \dots \diamond_s] = \mathsf{E}[\diamond_1, \dots, \diamond_{j-1}, \mathsf{D}[\diamond_j, \dots, \diamond_{j+m-1}], \diamond_{j+m}, \dots, \diamond_s]$ et où les $\mathtt{f}_i(b_i)$ sont égaux à $\mathtt{g}_i(u_i)$, $\mathtt{h}_{i-j+1}(d_{i+j-1})$ ou $\mathtt{g}_{i+1-m}(u_{i+1-m})$ suivant que $i$ soit dans l' intervalle $\{1, \dots, j-1\}$, dans l'intervalle $\{j, \dots, j+m-1\}$ ou dans l'intervalle $\{j+m, \dots, s\}$.
Ce critère demeure valide dans le cas où les appels $\mathtt{g}_i(\overline{u_i})$ correspondent à des appels de fonction qui quittent un appel récursif (i.e. des appels de fonction faisant appel à des définitions n'impliquant que des symboles de fonction strictement plus petits pour la précédence). Puisque nous considérons une valeur définie (i.e. une évaluation qui termine), de tels appels existent.
Définissons $P(\overline{X}) = \alpha \times Q_{\mathtt{f}}(\max(\overline{X}))$ avec $\alpha$ la constante du lemme II.4. Pour tout $i \in \{1, \dots, r\}$, $\mathtt{g}_i(\overline{u_i})$ termine et nous constatons que :

$$\begin{aligned}
P(\theta^*(\overline{u_i})) &\geq P(|\overline{u_i}|) && \text{Condition 3 de la définition II.5} \\
&\geq \alpha \times |[\![\mathtt{g}_i]\!](\overline{u_i})| && \text{Par construction de } Q_{\mathtt{f}} \\
&\geq \theta^*([\![\mathtt{g}_i]\!](\overline{u_i})) && \text{Lemme II.4}
\end{aligned}$$

Par conséquent, si $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{*} \mathsf{C}^*[\mathtt{g}_1(\overline{u_1}), \dots, \mathtt{g}_r(\overline{u_r})]$ alors :

$$\begin{aligned}
&\theta^*(\mathsf{C}^*)(P(\theta^*(\overline{u_1})), \dots, P(\theta^*(\overline{u_r}))) \\
&\geq \theta^*(\mathsf{C}^*)(\theta^*([\![\mathtt{g}_1]\!](\overline{u_1})), \dots, \theta^*([\![\mathtt{g}_r]\!](\overline{u_r}))) && \text{Par monotonie de } \theta^*(\mathsf{C}^*) \\
&\geq \|\mathtt{f}(v_1, \cdots, v_n)\| && \text{Corollaire II.7}
\end{aligned}$$

Il nous reste à démontrer qu'il existe une fonction $R_{\mathtt{f}} \in$ **Max-poly** $\{\mathbb{R}^+\}$ telle que $R_{\mathtt{f}}(\theta^*(\overline{v})) \geq \theta^*(\mathsf{C}^*)(P(\theta^*(\overline{u_1})), \dots, P(\theta^*(\overline{u_r})))$. L'inégalité (II.2) implique que

$\theta^*(\mathsf{C}^*)(\diamond_1, \cdots, \diamond_r)$ est polynomiale en $\diamond_j$ puisque $\theta^*(\mathsf{C}^*)$ est borné par un polynôme dépendant de $\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n))$ indépendamment de la longueur de dérivation. Nous appliquons encore le lemme II.4 et obtenons que $\|\mathtt{f}(v_1, \cdots, v_n)\|$ est borné polynomialement par $P'_{\mathtt{f}}(\max|v_i|) = R_{\mathtt{f}}(\alpha \times \max_{i=1..n}(|v_i|))$.

Pour conclure, $\forall \mathtt{f} \in Fct$, $P'_{\mathtt{f}} \in$ **Max-poly** $\{\mathbb{R}^+\}$, et nous pouvons aisément trouver un polynôme $P_{\mathtt{f}}$ tel que $\forall X\ P_{\mathtt{f}}(X) \geq P'_{\mathtt{f}}(X)$. □

**Corollaire II.11.** *Soit* $\boldsymbol{p}$ *un programme quasi-amical qui termine sur toutes ses entrées, alors pour tout symbole de fonction* $\mathtt{f}$ *d'arité* $n$ *dans* $\boldsymbol{p}$*, il existe un polynôme* $P_{\mathtt{f}}$ *tel que pour toute séquence de valeurs* $v_1, \cdots, v_n$ *:*

$$|[\![\mathtt{f}]\!](v_1, \cdots, v_n)| \leq P_{\mathtt{f}}(\max(|v_1|, \ldots, |v_n|))$$

**Exemple 19.** *On peut montrer que le programme décrit dans l'exemple 1 est quasi-amical en choisissant :*

$$\begin{aligned} \theta(\mathbf{S})(X) &= X + 1 \\ \theta(\mathtt{minus})(X, Y) &= X \end{aligned} \qquad \Bigg| \qquad \begin{aligned} \omega_{\mathtt{q}}(X, Y) &= X + Y \\ \omega_{\mathtt{minus}}(X, Y) &= X + Y \end{aligned}$$

*En effet, on vérifie aisément les conditions du critère quasi-amical :*

$$\begin{aligned} \omega_{\mathtt{minus}}(\theta^*(\mathbf{S}(v)), \theta^*(\mathbf{S}(u))) &= U + V + 2 \\ &\geq V + U \\ &= \omega_{\mathtt{minus}}(\theta^*(v), \theta^*(u)) \\ \omega_{\mathtt{q}}(\theta^*(\mathbf{S}(z)), \theta^*(\mathbf{S}(u))) &= U + Z + 2 \\ &= \theta^*(\mathbf{S})(\omega_{\mathtt{q}}(\theta^*(\mathtt{minus}(z, u)), \theta^*(\mathbf{S}(u)))) \end{aligned}$$

**Exemple 20** (Pgcd)**.** *Le programme suivant calcule le plus grand commun diviseur :*

$$\begin{aligned} &\mathtt{minus}(x, y) = \textbf{Case } x, y \textbf{ of} \\ &\qquad \mathbf{0}, z \to \mathbf{0} \\ &\qquad \mathbf{S}(z), \mathbf{0} \to \mathbf{S}(z) \\ &\qquad \mathbf{S}(u), \mathbf{S}(v) \to \mathtt{minus}(u, v) \\ &\mathtt{if}(x, y, z) = \textbf{Case } x, y, z \textbf{ of} \\ &\qquad \textbf{True}, u, v \to u \\ &\qquad \textbf{False}, u, v \to v \end{aligned}$$

$$\begin{aligned}
&\mathtt{gcd}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\\
&\quad \mathbf{0}, z \to z\\
&\quad \mathbf{S}(z), \mathbf{0} \to \mathbf{S}(z)\\
&\quad \mathbf{S}(u), \mathbf{S}(v) \to \mathtt{if}(\mathbf{le}(u,v), \mathtt{gcd}(\mathtt{minus}(v,u), \mathbf{S}(u)), \mathtt{gcd}(\mathtt{minus}(u,v), \mathbf{S}(v)))
\end{aligned}$$

**le** *est un opérateur spécifique qui, étant données deux entrées $n$ et $m$, retourne la valeur* **True** *(respectivement* **False***) si la représentation unaire de $n$ est plus petite (strictement plus grande) que celle de $m$. Par conséquent, $\theta(\mathbf{le})(X,Y) = 0$ définit bien une sup-interprétation du symbole* **le** *pour peu que l'on suppose que $\theta(\mathbf{True}) = \theta(\mathbf{False}) = 0$.*

*Ce programme possède deux fraternités :*

$$\mathtt{minus}(u,v)$$

*et*

$$\mathtt{if}(\mathbf{le}(u,v),\ \mathtt{gcd}(\mathtt{minus}\ (v,u), \mathbf{S}(u)), \mathtt{gcd}(\mathtt{minus}(u,v), \mathbf{S}(v)))$$

*La première dépend du symbole de fonction* `minus` *et nous laissons le lecteur vérifier qu'elle satisfait le critère quasi-amical. La deuxième est activée par $\mathtt{gcd}(\mathbf{S}(u), \mathbf{S}(v))$. En choisissant $\theta(\mathbf{S})(X) = X+1$, $\theta(\mathtt{if})(X,Y,Z) = \max(Y,Z)$ et $\theta(\mathtt{minus})(X,Y) = X$ comme sup-interprétations des symboles* **S**, `if` *et* `minus`*, il nous reste à vérifier qu'il existe un poids polynomial $\omega$ tel que :*

$$\begin{aligned}
&\omega_{\mathtt{gcd}}(U+1, V+1) = \omega_{\mathtt{gcd}}(\theta(\mathbf{S})(U), \theta(\mathbf{S})(V)) \geq\\
&\theta(\mathtt{if})(\theta(\mathbf{le})(U,V), \theta(\mathtt{minus})((V,U), \theta(\mathbf{S})(U)), \omega_{\mathtt{gcd}}(\theta(\mathtt{minus})(U,V), \mathbf{S}(V))) =\\
&\max(\omega_{\mathtt{gcd}}(V, U+1), \omega_{\mathtt{gcd}}(U, V+1))
\end{aligned}$$

*Il suffit de choisir $\omega_{\mathtt{gcd}}(X,Y) = X+Y$ et l'inégalité ci-dessus se réécrit en :*

$$U + V + 2 \geq V + U + 1$$

*Il en résulte que le programme est quasi-amical, et, nous pouvons donc appliquer le théorème II.10.*

**Exemple 21** (Codage de Huffman). *Le programme suivant correspond à l'algorithme des codes de Huffman décrit dans [BW88]. Le domaine de calcul est bâti à partir de deux symboles de constructeur pour les arbres,* **c** *pour les noeuds et* **tip** *pour les feuilles, et de trois symboles de constructeur* **0**, **1** *et* **nil** *d'arité 0 permettant d'encoder les mots binaires. Un mot binaire sera encodé de la manière suivante $\mathbf{c}(\mathbf{1}, \mathbf{c}(\mathbf{0}, \mathbf{nil}))$ et correspondra à un chemin relatif à un arbre donné.* **0** *correspond au fils gauche d'un noeud tandis que*

**1** *correspond à son fils droit. Nous commençons par la présentation de la fonction de décodage qui, pour un arbre t et un chemin p donnés, retourne le mot dans t correspondant au parcours du chemin p :*

$$\begin{aligned}
&\mathtt{decode}(t,p) = \mathtt{trace}(t,t,p)\\
&\mathtt{trace}(x,y,z) = \textbf{Case } x,y,z \textbf{ of}\\
&\qquad t,t',\mathbf{nil} \to \mathbf{nil}\\
&\qquad t,\mathbf{c}(\mathbf{Tip}(x),t_2),\mathbf{c}(\mathbf{0},p) \to \mathbf{c}(\mathbf{Tip}(x),\mathtt{trace}(t,t,p))\\
&\qquad t,\mathbf{c}(t_1,\mathbf{Tip}(x)),\mathbf{c}(\mathbf{1},p) \to \mathbf{c}(\mathbf{Tip}(x),\mathtt{trace}(t,t,p))\\
&\qquad t,\mathbf{c}(\mathbf{c}(t_1,t_2),\mathbf{c}(t_3,t_4)),\mathbf{c}(\mathbf{0},p) \to \mathtt{trace}(t,\mathbf{c}(t_1,t_2),p)\\
&\qquad t,\mathbf{c}(\mathbf{c}(t_1,t_2),\mathbf{c}(t_3,t_4)),\mathbf{c}(\mathbf{1},p) \to \mathtt{trace}(t,\mathbf{c}(t_3,t_4),p)
\end{aligned}$$

*Nous laissons au lecteur le soin de vérifier que le critère quasi-amical est vérifié en choisissant la sup-interprétation polynomiale et le poids polynomial suivants* $\theta(\mathbf{Tip})(X) = X+1$, $\theta(\mathbf{c})(X,Y) = X+Y+1$ *et* $\omega_{\mathtt{trace}}(X,Y,Z) = \max(X,Y)\times Z + \max(X,Y) + Z$.

*A présent, intéressons nous à la fonction de codage qui rend comme résultat le chemin dans l'arbre t correspondant à une liste de caractères p donnée en entrée :*

$$\begin{aligned}
&\mathtt{codes}(t,p) = \textbf{Case } t,p \textbf{ of}\\
&\qquad t,\mathbf{nil} \to \mathbf{nil}\\
&\qquad t,\mathbf{c}(x,y) \to \mathbf{c}(\mathtt{code}(t,x),\mathtt{codes}(t,y))\\
&\mathtt{code}(u,v) = \textbf{Case } u,v \textbf{ of}\\
&\qquad \mathbf{Tip}(x),y \to \mathtt{if}(x{=}y,\mathbf{nil},\mathbf{Err})\\
&\qquad \mathbf{c}(t_1,t_2),y \to \mathtt{if}(\mathtt{member}(y,t_1),\mathbf{c}(\mathbf{0},\mathtt{code}(t_1,y)),\\
&\qquad\qquad\qquad \mathtt{if}(\mathtt{member}(y,t_2),\mathbf{c}(\mathbf{1},\mathtt{code}(t_2,y)),\mathbf{Err}))\\
&\mathtt{member}(u,v) = \textbf{Case } u,v \textbf{ of}\\
&\qquad x,\mathbf{Tip}(y) \to \mathtt{if}(x{=}y,\mathbf{True},\mathbf{False})\\
&\qquad x,\mathbf{c}(t_1,t_2) \to \mathtt{or}(\mathtt{member}(x,t_1),\mathtt{member}(x,t_2))\\
&\mathtt{if}(u,v,w) = \textbf{Case } u,v,w \textbf{ of}\\
&\qquad \mathbf{True},x,y \to x\\
&\qquad \mathbf{False},x,y \to y
\end{aligned}$$

*Nous avons utilisé deux symboles d'opérateur dans ce programme.* $=$*, qui teste si deux caractères sont égaux, et* `or`*, qui calcule la disjonction logique. Puisque ces symboles ne retournent que des valeurs booléennes qui ont pour taille* 0*, nous pouvons leur attribuer*

*la sup-interprétation suivante* $\theta(=)(X,Y) = \theta(\texttt{or})(X,Y) = 0$*. Le symbole de fonction* `if` *est quasi-amical puisqu'il ne fait intervenir aucun appel récursif, et donc aucune fraternité, nous définissons sa sup-interprétation comme étant* $\theta(\texttt{if})(X,Y,Z) = \max(X,Y,Z)$*. Le symbole de fonction* `member` *calcule, lui aussi, des valeurs booléennes et nous pouvons définir* $\theta(\texttt{member})(X,Y) = 0$ *comme étant sa sup-interprétation. À présent, posons* $\theta(\mathbf{c})(X,Y) = X+Y+1$ *de telle sorte que la quasi-interprétation considérée soit additive. En choisissant* $\omega_{\texttt{code}}(X,Y) = X+Y$*, nous pouvons vérifier que* `code` *est quasi-amical. Puis, en choisissant* $\omega_{\texttt{codes}}(X,Y) = (X+1)\times(Y+1)$ *et* $\theta(\texttt{code})(T,X) = T$ *puisque l'expression* $\texttt{code}(t,x)$ *calcule le chemin de* $x$ *dans l'arbre* $t$*, nous vérifions aussi que* `codes` *est quasi-amical :*

$$\begin{aligned}
\omega_{\texttt{codes}}(\theta^*(t), \theta^*(\mathbf{c}(x,y))) &= (T+1)\times(X+Y+2) \\
&\geq (T+1)\times(Y+1)+T+1 \\
&= \theta(\mathbf{c})(\theta^*(\texttt{code}(t,x)), \omega_{\texttt{codes}}(\theta^*(t), \theta^*(y)))
\end{aligned}$$

*Il nous reste à étudier le programme qui construit l'arbre de Huffman. Étant donnée une liste de paires encodant un caractère et une densité, le programme commence par construire une liste de feuilles représentant les paires à l'aide du symbole de constructeur* **Tip** *(Dans le cas présent, le symbole de constructeur* **Tip** *a une arité* 2 *afin de pouvoir coder les caractères et les densités). Ensuite, l'algorithme fusionne les arbres de plus petite densité dans un nouvel arbre dont la densité est la somme des densités de ses noeuds. Finalement, le programme ordonne les arbres par densité croissante et s'appelle de manière récursive jusqu'à ce qu'il ne reste plus qu'un seul arbre, l'arbre d'Huffman. Ce programme est décrit par l'algorithme suivant où nous supposerons que la liste des noeuds, donnée en entrée, est déjà triée par densité croissante :*

$$\begin{aligned}
&\texttt{single}(u) = \mathbf{Case}\ u\ \mathbf{of} \\
&\quad \mathbf{nil} \to \mathbf{True} \\
&\quad \mathbf{c}(p, \mathbf{nil}) \to \mathbf{True} \\
&\quad \mathbf{c}(p, \mathbf{c}(q,l)) \to \mathbf{False} \\
&\texttt{head}(u) = \mathbf{Case}\ u\ \mathbf{of} \\
&\quad \mathbf{c}(p,q) \to p
\end{aligned}$$

$$
\begin{aligned}
&\texttt{weight}(u) = \textbf{Case } u \textbf{ of}\\
&\quad \textbf{Tip}(x,w) \to w\\
&\quad \textbf{c}(t_1,t_2) \to \texttt{add}(\texttt{weight}(t_1),\texttt{weight}(t_2))\\
&\texttt{tiping}(u) = \textbf{Case } u \textbf{ of}\\
&\quad \textbf{nil} \to \textbf{nil}\\
&\quad \textbf{c}((x,w),p) \to \textbf{c}(\textbf{Tip}((x,w)),\texttt{tiping}(p))
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{insert}(u,v) = \textbf{Case } u,v \textbf{ of}\\
&\quad p,\textbf{nil} \to \textbf{c}(p,\textbf{nil})\\
&\quad p,\textbf{c}(q,r) \to \texttt{if}(\textbf{le}(\texttt{weight}(p),\texttt{weight}(q)),\textbf{c}(p,\textbf{c}(q,r)),\textbf{c}(q,\texttt{insert}(p,r)))\\
&\texttt{combine}(u) = \textbf{Case } u \textbf{ of}\\
&\quad p \to \texttt{if}(\texttt{single}(p),\texttt{head}(p),\texttt{combine}(p))\\
&\quad \textbf{c}(p,\textbf{c}(q,l)) \to \texttt{insert}(\textbf{c}(p,q),l)\\
&\texttt{build}(p) = \texttt{combine}(\texttt{tiping}(p))
\end{aligned}
$$

*On vérifiera que le programme est quasi-amical en prenant :*

$$
\begin{aligned}
&\omega_{\texttt{combine}}(X) = \omega_{\texttt{tiping}}(X) = \omega_{\texttt{weight}}(X) = X \text{ et } \omega_{\texttt{insert}}(X,Y) = X+Y\\
&\theta(\texttt{weight})(X) = \theta(\texttt{head})(X) = X,\ \theta(\texttt{add})(X,Y) = X+Y \text{ et } \theta(\texttt{single})(X) = 0
\end{aligned}
$$

**Exemple 22.** *Le programme de l'exemple 17 n'est pas quasi-amical. En effet, la sup-interprétation de* `double` *doit être plus grande que* $2 \times X$ *et, si l'on veut vérifier le critère quasi-amical, il faut trouver un poids polynomial* $\omega_{\exp}$ *tel que :*

$$\omega_{\exp}(X+1) \geq \theta(\texttt{double})(\omega_{\exp}(X)) \geq 2 \times \omega_{\exp}(X)$$

*ce qui est clairement impossible.*

### II.2.2 Critère pour les programmes non-terminants

Nous introduisons ici un nouveau critère afin de renforcer le résultat du théorème II.10. Ce critère garantit que, même en cas de non terminaison du programme, la taille de toutes les valeurs intermédiaires calculées et, par conséquent, la taille de tout stack frame est bornée polynomialement par la taille des entrées. L'objectif d'un tel critère est de permettre le contrôle de la taille des sorties d'un programme potentiellement non terminant. En outre, il va permettre de considérer des programmes sur des types de données infinies comme les streams. On pourrait donc envisager de l'étendre à la programmation réactive comme dans [ADZ04].

**Définition II.12** (Appels récursifs bornés)**.** *Un programme* $\boldsymbol{p}$ *possède des* appels récursifs bornés *si et seulement s'il admet une sup-interprétation polynomiale et additive* $\theta$ *et un poids polynomial* $\omega$ *tels que pour toute fraternité de la forme* $\mathsf{C}[\mathtt{g}_1(\overline{e_1}),\ \ldots, \mathtt{g}_r(\overline{e_r})]$*, activée par* $\mathtt{f}(p_1, \cdots, p_n)$*, on ait :*

$$\omega_{\mathtt{f}}(\theta^*(p_1), \ldots, \theta^*(p_n)) \geq \max_{i=1..r}(\omega_{\mathtt{g}_i}(\theta^*(\overline{e_i})))$$

Notons que le critère à appels récursifs bornés est indépendant du critère quasi-amical de la section précédente comme l'illustrent les exemples qui suivent :

**Exemple 23.**

$$\begin{aligned}\mathtt{half}(t) = \mathbf{Case}\ t\ \mathbf{of}\ \ &\\ &\mathbf{S}(\mathbf{S}(x)) \rightarrow \mathbf{S}(\mathtt{half}(x))\\ &\mathbf{S}(\mathbf{0}) \rightarrow \mathbf{0}\\ &\mathbf{0} \rightarrow \mathbf{0}\\ \mathtt{f}(x) = \mathtt{half}(\mathtt{f}(&\mathtt{double}(x)))\end{aligned}$$

*où* `double` *est la fonction décrite dans l'exemple 17. La taille des arguments de* `f` *est doublée à chaque application de la dernière définition. Cependant, en prenant* $\theta(\mathtt{half})(X) = X/2$*,* $\theta(\mathtt{double})(X) = 2 \times X$ *et* $\omega_{\mathtt{f}}(X) = X$*, on peut vérifier que le critère quasi-amical est satisfait, alors que le programme n'est pas à appels récursifs bornés. En effet, il faudrait trouver un poids polynomial* $\omega_{\mathtt{f}}$ *vérifiant* $\omega_{\mathtt{f}}(X) \geq \omega_{\mathtt{f}}(2 \times X)$*, ce qui est impossible.*

**Exemple 24.** *Le programme de l'exemple 9 :*

$$\begin{aligned}\mathtt{double}(y) = \mathbf{Case}\ y\ \mathbf{of}\ &\mathbf{S}(x) \rightarrow \mathbf{S}(\mathbf{S}(\mathtt{double}(x)))\\ &\mathbf{0} \rightarrow \mathbf{0}\\ \exp(y) = \mathbf{Case}\ y\ \mathbf{of}\ &\mathbf{S}(x) \rightarrow \mathtt{double}(\exp(x))\\ &\mathbf{0} \rightarrow \mathbf{S}(\mathbf{0})\end{aligned}$$

*est à appels récursifs bornés mais n'est pas quasi-amical. En effet, un programme calculant une fonction exponentielle ne saurait vérifier le critère quasi-amical. En revanche, en prenant* $\theta(\mathbf{S})(X) = X + 1$ *et* $\omega_{\exp}(X) = X$*, on vérifie aisément que* $\omega_{\exp}(\theta^*(\mathbf{S}(x))) \geq \omega_{\exp}(\theta^*(x))$*.*

**Lemme II.13.** *Si un programme à appels récursifs bornés a un arbre des appels contenant une branche de la forme* $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, u_1, \cdots, u_m\rangle$ *avec* $\mathtt{f} \approx_{Fct} \mathtt{g}$ *alors :*

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m))$$

*Démonstration.* Nous montrons ce résultat par induction sur le nombre $k$ d'états de la branche :

- Si $k = 1$, $\langle \mathtt{f}, v_1, \cdots, v_n \rangle \rightsquigarrow \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ alors il existe une définition contenant une fraternité $\mathsf{C}[\ldots, \mathtt{g}(e_1, \cdots, e_m), \ldots]$, activée par $\mathtt{f}(p_1, \cdots, p_n)$ avec $\mathtt{f} \approx_{Fct} \mathtt{g}$, et il existe une substitution $\sigma$ telle que $p_i\sigma = v_i$ et $[\![e_j\sigma]\!] = u_j$. En combinant la condition du critère à appels récursifs bornés, la monotonie des poids et le lemme II.6, on obtient :

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \omega_{\mathtt{g}}(\theta^*(e_1\sigma), \cdots, \theta^*(e_m\sigma)) \geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m))$$

- Supposons, par hypothèse d'induction, que si $\langle \mathtt{f}, v_1, \cdots, v_n \rangle \overset{l}{\rightsquigarrow} \langle \mathtt{h}, v'_1, \cdots, v'_r \rangle$ avec $\mathtt{f} \approx_{Fct} \mathtt{h}$ et $l \leq k$, alors on a

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \omega_{\mathtt{h}}(\theta^*(v'_1), \cdots, \theta^*(v'_r)) \quad (H.I.)$$

  Maintenant considérons la branche suivante de l'arbre des appels de longueur $k+1$, avec $\mathtt{g} \approx_{Fct} \mathtt{f}$ :

$$\langle \mathtt{f}, v_1, \cdots, v_n \rangle \overset{k}{\rightsquigarrow} \langle \mathtt{h}, v'_1, \cdots, v'_r \rangle \rightsquigarrow \langle \mathtt{g}, u_1, \cdots, u_m \rangle$$

$$\begin{aligned} \omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) &\geq \omega_{\mathtt{h}}(\theta^*(v'_1), \cdots, \theta^*(v'_r)) && \text{Par H.I.} \\ &\geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m)) && \text{Par H.I.} \end{aligned}$$

□

**Théorème II.14.** *Étant donné un programme* **p** *quasi-amical à appels récursifs bornés. Pour tout symbole de fonction* $\mathtt{f}$ *de* **p***, il existe un polynôme* $R_{\mathtt{f}}$ *tel que, pour tout état* $\langle \mathtt{g}, u_1, \cdots, u_m \rangle$ *de l'arbre d'appel de racine* $\langle \mathtt{f}, v_1, \cdots, v_n \rangle$*, on ait :*

$$\max_{i=1..m}(|u_i|) \leq R_{\mathtt{f}}(\max(|v_1|, ..., |v_n|))$$

*même si* $\mathtt{f}(v_1, \ldots, v_n)$ *ne termine pas.*

*Démonstration.* Étant donné un arbre des appels de racine $\langle \mathtt{f}, v_1, \cdots, v_n \rangle$ correspondant à une exécution d'un programme **p**. On définit le niveau d'un symbole de fonction par $\mathtt{lv}(\mathtt{f}) =_{\text{def}} 0$, si $\mathtt{f}$ est la fonction correspondant à la racine de l'arbre des appels. Si $\mathtt{h} \approx_{Fct} \mathtt{g}$ alors $\mathtt{lv}(\mathtt{h}) =_{\text{def}} \mathtt{lv}(\mathtt{g})$. Pour tout $\mathtt{g}$ tel que $\mathtt{g} >_{Fct} \mathtt{h}$, $\mathtt{lv}(\mathtt{h}) =_{\text{def}} \max_{\mathtt{g} >_{Fct} \mathtt{h}}(\mathtt{lv}(\mathtt{g})) + 1$. Il est important de constater que le niveau est borné par la taille du programme puisqu'il dépend du nombre de symboles de fonction. On étend la notion de niveau aux états de l'arbre des appels : Le niveau d'un état est le niveau du symbole de fonction

correspondant. Le niveau d'un arbre des appels est le plus haut niveau d'un symbole de fonction apparaissant dans un des états de l'arbre. À présent, nous supposerons que l'arbre des appels considéré dans cette preuve est de niveau $\gamma$. Nous allons construire le polynôme $R$ par induction sur le niveau. Soit $\langle \mathtt{g}, u_1, \cdots, u_m \rangle$ un état de l'arbre des appels,

- Si $\mathtt{lv}(\mathtt{g}) = 0$ alors $\mathtt{f} \approx_{Fct} \mathtt{g}$. Posons $R_0(X) = \omega_{\mathtt{f}}(\alpha \times X, \ldots, \alpha \times X)$, $\alpha$ étant la constante du lemme II.4. On vérifie que :

$$\begin{aligned} R_0(\max_{j=1..n}(|v_j|)) &\geq \omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) && \text{Lemme II.4} \\ &\geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m)) && \text{Lemme II.13} \\ &\geq \max_{i=1..m}(\theta^*(u_i)) && \text{Condition 2 de la définition II.8} \\ &\geq \max_{i=1..m}(|u_i|) && \text{Condition 3 de la définition II.5} \end{aligned}$$

- Si $\mathtt{lv}(\langle \mathtt{g}, u_1, \cdots, u_m \rangle) = k+1$. Supposons que nous ayons déjà construit un polynôme $R_k$ pour le niveau $k$. En considérant la branche de l'arbre des appels allant de $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$ à $\langle \mathtt{g}, u_1, \cdots, u_m \rangle$, on sait qu'il existe deux états $\langle \mathtt{h}, v'_1, \cdots, v'_l \rangle$ et $\langle \mathtt{h'}, u'_1, \cdots, u'_j \rangle$ de niveau respectif $k$ et $k+1$ vérifiant :

$$\begin{array}{lllllll} \langle \mathtt{f}, v_1, \cdots, v_n \rangle & \overset{*}{\rightsquigarrow} & \langle \mathtt{h}, v'_1, \cdots, v'_l \rangle & \rightsquigarrow & \langle \mathtt{h'}, u'_1, \cdots, u'_j \rangle & \overset{*}{\rightsquigarrow} & \langle \mathtt{g}, u_1, \cdots, u_m \rangle \\ 0 & & k & & k+1 & & k+1 \end{array}$$

En outre, il existe une substitution $\sigma$ et une définition de la forme $\mathtt{h}(x_1, \cdots, x_l) =$ **Case** $x_1, \cdots, x_l$ **of** $p_1, \cdots, p_l \rightarrow \mathsf{C}[\mathtt{h'}(e_1, \cdots, e_j)]$ telles que $p_i\sigma = v'_i$ et $[\![e_i\sigma]\!] = u'_i$. Le programme étant quasi-amical, en appliquant le théorème II.10, on peut trouver un polynôme $P_{e_i}$ vérifiant $P_{e_i}(\max_{i=1..l}(|v'_i|)) \geq \max_{i=1..j}(|u'_i|)$. On pose :

$$\begin{aligned} Q_{def}(X) &= \max_{i=1..j} P_{e_i}(X) \\ \text{et } S^k_{def}(X) &= S(\alpha \times Q_{def}(R_k(X))) \end{aligned}$$

avec $S(X) = \omega_{\mathtt{h'}}(X, \ldots, X)$ et $\alpha$ la constante du lemme II.4. Intuitivement, $S^k_{def}$ représente une borne supérieure polynomiale sur la taille des valeurs dans les états de niveau inférieur ou égal à $k+1$ dans la branche considérée de l'arbre des appels. Par Hypothèse d'Induction (H.I.), $R_k$ doit vérifier $R_k(\max_{i=1..n}|v_i|) \geq \max_{i=1..l}|v'_i|$.

On déduit de ceci que :

$$
\begin{aligned}
S^k_{def}(\max_{i=1..n} |v_i|) &= S(\alpha \times Q_{def}(R_k(\max_{i=1..n} |v_i|))) && \\
&\geq S(\alpha \times Q_{def}(\max_{i=1..l} |v'_i|)) && \text{H.I. et monotonie de } Q_{def} \\
&\geq S(\max_{i=1..j}(\alpha \times |u'_i|)) && \text{Par définition de } Q_{def} \\
&\geq S(\max_{i=1..j}(\theta^*(u'_i))) && \text{Lemme II.4} \\
&\geq \omega_{\mathtt{h'}}(\theta^*(u'_1), \cdots, \theta^*(u'_j)) && \text{Par monotonie du poids} \\
&\geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m)) && \text{Lemme II.13} \\
&\geq \max_{i=1..m}(|u_i|) && \text{Définition II.8}
\end{aligned}
$$

Maintenant, il nous faut construire une borne polynomiale sur toutes les valeurs des états de niveau inférieur ou égal à $k+1$ et pas seulement ceux de la branche de l'arbre d'appel considérée ci-dessus. À cet effet, soit $E_k$ l'ensemble des définitions de la forme $\mathtt{h}(x_1, \cdots, x_l)$ = **Case** $x_1, \cdots, x_l$ **of** $p_1, \cdots, p_l \rightarrow$ $\mathtt{C}[\mathtt{h'}(e_1, \cdots, e_j)]$, `h` et `h'` ayant pour niveau respectif $k$ et $k+1$. Comme dans le précédent cas de figure, on définit les polynômes $Q_{def}$ et $S^k_{def}$ pour toute définition $def \in E_k$. Il reste à définir $R_{k+1}(X) = \max_{def \in E_k}(S^k_{def}(X))$

Par construction, $R_{k+1}$ représente un borne polynomiale sur la taille des valeurs contenues dans des états de niveau inférieur ou égal à $k+1$. On choisit alors $R_{\mathtt{f}}$ comme étant le polynôme qui vérifie $R_{\mathtt{f}}(X) \geq R_\gamma(X)$ avec $\gamma$ le niveau du programme, $R_{\mathtt{f}}$ correspond alors à la borne désirée sur toute valeur contenue dans un état de l'arbre d'appel. Il est nécessaire de souligner qu'un tel polynôme $R_{\mathtt{f}}$ existe puisque $R_\gamma$ est une fonction de **Max-poly** $\{\mathbb{R}\}$, le nombre de compositions de polynômes restant borné à chaque étape de la preuve par $\gamma$, et $\gamma$ étant lui-même borné par la taille du programme. □

**Exemple 25** (Streams)**.** *Comme nous l'avons fait remarquer dans l'introduction de cette section, le théorème II.14 trouve tout son intérêt lorsqu'il est appliqué à des programmes ne terminant pas. En particulier, il reste valide pour une classe de programmes incluant des streams. Dans cette optique, nous introduisons des données de type stream à notre langage de programmation à l'aide d'un symbole binaire de constructeur « :: ». Dans un stream $h :: t$, $h$ est appelé la tête du stream et $t$ est sa queue. On suppose qu'une sémantique sur les streams est définie de manière classique en utilisant une évaluation*

*paresseuse sur les streams [FM03].*

$$\begin{aligned}
&\texttt{add}(x,y) = \textbf{Case } x, y \textbf{ of}\\
&\qquad \mathbf{S}(u), v \to \mathbf{S}(\texttt{add}(u,v))\\
&\qquad \mathbf{0}, v \to v\\
&\texttt{addstream}(x,y) = \textbf{Case } x, y \textbf{ of } u :: l, v :: l' \to \texttt{add}(u,v) :: \texttt{addstream}(l,l')
\end{aligned}$$

*Ce programme, qui fusionne deux streams, est quasi-amical à appels récursifs bornés en prenant* $\theta(\texttt{add})(X,Y) = X+Y$, $\theta^*(x :: l) = X+L+1$, $\omega_{\texttt{add}}(X,Y) = X+Y$ *et* $\omega_{\texttt{addstream}}(X,Y) = X+Y$ *:*

- *La condition du critère quasi-amicale est satisfaite :*

$$\begin{aligned}
\omega_{\texttt{add}}(\theta^*(\mathbf{S}(u)), \theta^*(v)) &= U+V+1\\
&\geq U+V+1 = \theta(\mathbf{S})(\omega_{\texttt{add}}(\theta^*(u), \theta^*(v)))\\
\omega_{\texttt{addstream}}(\theta^*(u :: l), \theta^*(v :: l')) &= U+V+L+L'+2\\
&\geq L+L'+U+V+1\\
&= \theta(::)(\theta^*(\texttt{add}(u,v)), \omega_{\texttt{addstream}}(\theta^*(l), \theta^*(l')))
\end{aligned}$$

- *La condition du critère à appels récursifs bornés est, elle aussi, satisfaite :*

$$\begin{aligned}
\omega_{\texttt{add}}(\theta^*(\mathbf{S}(u)), \theta^*(v)) &= U+V+1\\
&\geq U+V = \omega_{\texttt{add}}(\theta^*(u), \theta^*(v))\\
\omega_{\texttt{addstream}}(\theta^*(u :: l), \theta^*(v :: l')) &= U+V+L+L'+2\\
&\geq L+L' = \omega_{\texttt{addstream}}(\theta^*(l), \theta^*(l'))
\end{aligned}$$

*Ainsi, le théorème II.14 peut être appliqué. Ceci étant, il serait inutile de considérer le stream dans son entier comme étant une entrée, puisque la taille d'un stream est non bornée. Pour palier à ce problème, on choisit les entrées comme étant un nombre restreint de têtes de stream, nombre fixé par la syntaxe du programme. De la même manière, tout programme d'application sur les streams de la forme :*

$$\texttt{f}(x) = \textbf{Case } x \textbf{ of } z :: l \to \texttt{g}(z) :: \texttt{f}(l)$$

*est quasi-amical à appels récursifs bornés du moment que le symbole* g *représente un programme quasi-amical. Le théorème II.14 s'applique donc encore. De plus, pour tous ces programmes nous savons que les valeurs calculées dans le stream de sortie (i.e. dans les têtes situées dans le membre droit d'une définition) ont une taille bornée polynomialement par la taille de quelques têtes d'entrée, dont le nombre est fixé par la taille du*

*programme, puisque les calculs impliquent uniquement des symboles de fonction quasi-amicaux à valeur sur des données finies (sinon certaines parties du programme ne sont jamais évaluées, et dans ce cas, une borne sur leurs calculs est inutile). Pour finir, nous présentons un exemple de programme ne vérifiant pas le critère quasi-amical à appels récursifs bornés :*

$$\mathtt{f}(x) = \textbf{Case } x \textbf{ of } z :: l \rightarrow \mathtt{f}(z :: z :: l)$$

*Ce programme ne remplit pas les conditions du critère puisqu'il empile infiniment des têtes de stream sur son argument de récurrence, calculant de cette manière une valeur de taille non bornée. Pour vérifier le critère, en supposant que* $\theta(::)(X,Y) = X + Y + 1$*, il nous faudrait trouver un poids polynomial* $\omega_{\mathtt{f}}$ *vérifiant :*

$$\omega_{\mathtt{f}}(Z + L + 1) \geq \omega_{\mathtt{f}}(2 \times Z + L + 2)$$

*Ce qui est clairement impossible.*

### II.2.3 Critère pour les algorithmes de type « diviser pour régner »

Lorsque l'on cherche à contrôler la taille des valeurs calculées par un appel récursif, il arrive que l'on soit obligé de connaître une borne supérieure précise sur la taille des arguments de la récurrence pour être capable de la contrôler. Cependant, l'obtention d'une précision raisonnable peut s'avérer être une tâche particulièrement ténue pour peu qu'un programme utilise des opérations spécifiques de destruction sur les arguments de ses appels récursifs, comme l'illustre l'exemple qui suit :

**Exemple 26** (Identité)**.**

$$\begin{aligned} \mathtt{f}(x) = \textbf{Case } & x \textbf{ of} \\ & l \rightarrow \mathbf{c}(\mathbf{hd}(l), \mathtt{f}(\mathbf{tl}(l))) \\ & \mathbf{nil} \rightarrow \mathbf{nil} \end{aligned}$$

*Ce programme calcule la fonction identité sur les listes en utilisant des opérateurs destructeurs* **tl** *et* **hd***. Ces deux destructeurs calculent respectivement la queue et la tête d'une liste* $l$ *donnée en entrée. Malheureusement, leurs sup-interprétations* $\theta(\mathbf{tl})(L)$ *et* $\theta(\mathbf{hd})(L)$ *doivent être au moins égales à* $L$ *afin de borner la taille des valeurs calculées. Si nous cherchons à vérifier le critère quasi-amical en prenant* $\theta(\mathbf{c})(X,Y) = X + Y + k$*, avec* $k$ *une constante supérieure ou égale à* 1*, alors on obtient :*

$$\omega_{\mathtt{f}}(L) \geq L + k + \omega_{\mathtt{f}}(L)$$

*Il en découle que le programme n'est pas quasi-amical.*

Ce problème résulte du fait que $\theta(\mathbf{hd})(L)$ et $\theta(\mathbf{tl})(L)$ soient considérées comme des fonctions de $L$, générant ainsi une borne supérieur trop large. Dans le cas présent, l'opération de destruction oblige la sup-interprétation à être sous-terme (i.e. $\theta(\mathbf{hd})(L) \geq L$), ce qui s'avère être un inconvénient intensionnel rédhibitoire. Historiquement, les sup-interprétations ont été introduites afin d'éviter d'avoir recours à la propriété sous-terme qui était le principal inconvénient des quasi-interprétations. Ce sujet sera débattu plus largement dans la section II.3.1 et nous invitons le lecteur intéressé à s'y reporter directement. Dans cette section, nous essayons d'endiguer ce problème, de manière syntaxique, en remplaçant la sup-interprétation d'un symbole de destructeur par des nouvelles variables qui satisfont un système de contraintes.

**Définition II.15** (Projecteur)**.** *Un symbole de fonction $\mathtt{d}_i^{\mathbf{c}}$ est appelé le i-ème projecteur relatif au symbole de constructeur* $\mathbf{c}$ *s'il est défini par :*

$$\mathtt{d}_i^{\mathbf{c}}(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{c}(e_1, \cdots, e_n) \to e_i$$

**Remarque 4.** *La notion de projecteur introduite ci-dessus est très restrictive d'un point de vue syntaxique. Cependant, elle a le mérite de proposer une approche simple permettant de traiter ce type de problème. On pourrait envisager d'ajouter à un programme des opérateurs destructeurs beaucoup plus sophistiqués. On augmente alors considérablement le nombre de programmes capturés mais, en contrepartie, on augmente aussi l'indécidabilité de la méthode, puisque trouver la sup-interprétation de tels symboles peut s'avérer extrêmement difficile, voire impossible. Ceci étant, une autre approche raisonnable peut être de considérer les opérateurs comme des opérations de base du langage, fournies avec un certificat, leur sup-interprétation.*

La sup-interprétation d'un projecteur $\mathtt{d}_j^{\mathbf{c}}(e)$ est alors définie comme étant une nouvelle variable.

**Définition II.16** (Sup-interprétation de projecteur)**.** *Étant donné un projecteur $\mathtt{d}_j^{\mathbf{c}}$, une expression $e$ et une sup-interprétation $\theta$, l'extension canonique $\theta^*$ de $\theta$ définie sur une expression $\mathtt{d}_j^{\mathbf{c}}(e)$ est modifiée comme suit :*

$$\theta^*(\mathtt{d}_j^{\mathbf{c}}(e)) =_{def} X^e_{\mathtt{d}_j^{\mathbf{c}}}$$

*où $X^e_{\mathtt{d}_j^{\mathbf{c}}}$ est une nouvelle variable.*

Pour tout programme, en présence de projecteurs, on génère alors un ensemble de contraintes :

$$S := \emptyset$$

Pour toute expression $\mathtt{d}^{\mathbf{c}}_j(e)$ de $\mathbf{p}$ avec $\mathtt{d}^{\mathbf{c}}_j$ projecteur, $S := S \cup \left\{ \sum_{j=1}^{n} X^{e}_{\mathtt{d}^{\mathbf{c}}_j} + 1 \leq \theta^*(e) \right\}$

Fig. II.3: Génération de contraintes sur les projecteurs

**Définition II.17** (Contraintes sur les projecteurs)**.** *Soit $\mathbf{p}$ un programme, on génère l'ensemble $S$ des contraintes sur les projecteurs dans la figure II.3.*

Ces inégalités correspondent à des contraintes sur les sup-interprétations d'expressions ayant été projetées. En pratique, ces contraintes sont toujours satisfaites pour des sup-interprétations additives. En effet, supposons que $e = \mathbf{c}(e_1, \cdots, e_n)$. On a :

$$\begin{aligned}\theta^*(e) &= \theta(\mathbf{c})(\theta^*(e_1), \cdots, \theta^*(e_n)) \\ &= \theta(\mathbf{c})(\theta^*(\mathtt{d}^{\mathbf{c}}_1(e), \ldots, \mathtt{d}^{\mathbf{c}}_n(e))) \\ &= \theta(\mathbf{c})(X^{e}_{\mathtt{d}^{\mathbf{c}}_1}, \ldots, X^{e}_{\mathtt{d}^{\mathbf{c}}_n})\end{aligned}$$

Par conséquent, les contraintes sur les projecteurs sont de la forme $\sum_{j=1}^{n} X^{e}_{\mathtt{d}^{\mathbf{c}}_j} + 1 \leq \theta(\mathbf{c})(X^{e}_{\mathtt{d}^{\mathbf{c}}_1}, \ldots, X^{e}_{\mathtt{d}^{\mathbf{c}}_n})$ et sont toujours vérifiées puisque $\theta(\mathbf{c})(\diamond_1, \cdots, \diamond_n) = \sum_{i=1}^{n} \diamond_i + k$ avec $k \geq 1$.

**Définition II.18** (Quasi-amical modulo projection)**.** *Étant donnés un programme $\mathbf{p}$ et $S$ l'ensemble de ses contraintes sur les projecteurs, $\mathbf{p}$ est quasi-amical modulo projection s'il existe une sup-interprétation polynomiale et additive $\theta$ et un poids polynomial $\omega$ tels que « $S$ implique que $\mathbf{p}$ est quasi-amical ».*

**Exemple 27.** *Considérons le programme suivant qui renverse une liste donnée en entrée :*

$$\begin{aligned}\mathtt{reverse}(l) &= \mathtt{rev}(l, \mathbf{nil}) \\ \mathtt{rev}(l, a) &= \mathtt{if}(l{=}\mathbf{nil}, a, \mathtt{rev}(\mathbf{tl}(l), \mathbf{c}(\mathbf{hd}(l), a)))\end{aligned}$$

*Le système de contraintes sur les projecteurs généré est le suivant :*

$$S = \left\{ X^{\mathbf{tl}(l)} + X^{\mathbf{hd}(l)} + 1 \leq L) \right\}$$

*Ce programme possède une seule fraternité $\mathtt{rev}(\mathbf{tl}(l), \mathbf{c}(\mathbf{hd}(l), a))$. De sorte que le critère quasi-amical engendre l'inégalité qui suit :*

$$\omega_{\mathtt{rev}}(L, A) \geq \omega_{\mathtt{rev}}(X^{\mathbf{tl}(l)}, A + X^{\mathbf{hd}(l)} + k)$$

*où $k$ est une constante additive associé au symbole de constructeur de listes* **c**. *En choisissant $k = 1$ et $\omega_{\texttt{rev}}(X, Y) = X + Y$, on obtient :*

$$L + A \geq X^{\mathbf{tl}(l)} + X^{\mathbf{hd}(l)} + A + 1$$

*Finalement, $S$ implique que* **p** *est quasi-amical et le programme est quasi-amical modulo projection.*

**Exemple 28.** *Le programme de l'exemple 26 est quasi-amical modulo projection. En effet, il est suffisant de prendre $\omega_{\texttt{f}}(X) = X$ et $\theta(\mathbf{c})(X, Y) = X + Y + 1$ pour le prouver.*

**Théorème II.19.** *Soit* **p** *un programme quasi-amical modulo projection. Pour tout symbole de fonction* `f` *d'arité $n$ de* **p**, *il existe un polynôme $P_{\texttt{f}}$ qui vérifie pour toute séquence de valeurs $v_1, \ldots, v_n$,*

$$\|\texttt{f}(v_1, \ldots, v_n)\| \leq P_{\texttt{f}}(\max(|v_1|, ..., |v_n|))$$

*Démonstration.* Le système $S$ est toujours satisfait, comme nous l'avons fait observer ci-dessus. La satisfaction de la proposition « $S$ implique que **p** est quasi-amical »est donc équivalente à la satisfaction de la proposition « **p** est quasi-amical ». □

**Exemple 29** (Quicksort)**.** *Le programme suivant calcule l'algorithme Quicksort en utilisant un symbole de fonction intermédiaire* `order`. *Étant donnés un nombre unaire $n$ et une liste $l$, ce symbole calcule une paire* **pair**$(u, v)$ *de deux listes $u$ et $v$ qui représentent respectivement les éléments de la liste d'entrée $l$ inférieurs ou égaux à $n$ et les éléments de $l$ strictement plus grands que $n$.*

$$\begin{aligned}
&\texttt{append}(x, y) = \mathbf{Case}\ x, y\ \mathbf{of}\\
&\qquad \mathbf{nil}, u \to u\\
&\qquad \mathbf{c}(n, v), u \to \mathbf{c}(n, \texttt{append}(v, u))\\
&\texttt{p}_1(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{pair}(p_1, p_2) \to p_1\\
&\texttt{p}_2(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{pair}(p_1, p_2) \to p_2\\
&\texttt{order}(w, x, y, z) = \mathbf{Case}\ w, x, y, z\ \mathbf{of}\\
&\qquad n, \mathbf{c}(m, l), u, v \to \texttt{if}(\mathbf{le}(m, n), \texttt{order}(n, l, \mathbf{c}(m, u), v), \texttt{order}(n, l, u, \mathbf{c}(m, v))\\
&\qquad n, \mathbf{nil}, u, v \to \mathbf{pair}(u, v)\\
&\texttt{qs}(x) = \mathbf{Case}\ x\ \mathbf{of}\\
&\mathbf{nil} \to \mathbf{nil}\\
&\mathbf{c}(n, u) \to \texttt{append}(\texttt{qs}(\texttt{p}_1(\texttt{order}(n, u, \mathbf{nil}, \mathbf{nil}))), \mathbf{c}(n, \texttt{qs}(\texttt{p}_2(\texttt{order}(n, u, \mathbf{nil}, \mathbf{nil})))))
\end{aligned}$$

$\texttt{append}$ *est un programme quasi-amical pour peu que l'on prenne* $\theta(\mathbf{c})(X,Y) = X+Y+1$, $\theta(\mathbf{S})(X) = X+1$ *et* $\omega_{\texttt{append}}(X,Y) = X+Y$. *Puisque* $\texttt{p}_1$ *et* $\texttt{p}_2$ *sont des projecteurs, l'ensemble* $S$ *des contraintes de projecteurs est égal à :*

$$\left\{X_{\texttt{p}_1}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})} + X_{\texttt{p}_2}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})} + 1 \leq \theta^*(\texttt{order}(n,u,\mathbf{nil},\mathbf{nil}))\right\}$$

*De plus, en posant* $\theta(\texttt{order})(W,X,Y,Z) = X+Y+Z+1$, $S$ *se réécrit en :*

$$\left\{X_{\texttt{p}_1}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})} + X_{\texttt{p}_2}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})} \leq U\right\}$$

*En choisissant* $\theta(\texttt{if})(X,Y,Z) = \max(Y,Z)$ *et* $\theta(\texttt{append})(X,Y) = X+Y$, *il nous reste à démontrer que :*

$$\begin{aligned}\omega_{\texttt{order}}(N,M+L+1,U,V) &\geq \omega_{\texttt{order}}(N,L,M+U+1,V)\\ &\geq \omega_{\texttt{order}}(N,L,U,V+M+1)\\ \omega_{\texttt{qs}}(N+U+1) &\geq \sum_{i=1}^{2}\omega_{\texttt{qs}}(X_{\texttt{p}_i}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})}) + N + 1\end{aligned}$$

*afin de prouver que le programme est quasi-amical. En choisissant* $\omega_{\texttt{qs}}(X) = X$ *et* $\omega_{\texttt{order}}(W,X,Y,Z) = W+X+Y+Z$, *le système d'inégalités ci-dessus se réécrit en :*

$$\begin{aligned}&\{N+M+L+U+V+1 \geq N+M+L+U+V+1,\\ &N+U+1 \geq \max(X_{\texttt{p}_1}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})}, X_{\texttt{p}_2}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})}),\\ &N+U+1 \geq X_{\texttt{p}_1}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})} + X_{\texttt{p}_2}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})} + N + 1\ \}\end{aligned}$$

*qui est égal à* $\left\{U \geq X_{\texttt{p}_1}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})} + X_{\texttt{p}_2}^{\texttt{order}(n,u,\mathbf{nil},\mathbf{nil})}\right\} = S$, *de telle sorte que le programme est quasi-amical modulo projection.*

## II.3 Application aux travaux précédents

### II.3.1 Comparaison avec les quasi-interprétations

Une quasi-interprétation (Cf. définition I.13), tout comme une sup-interprétation, représente une assignation qui fournit une borne supérieure sur la taille des valeurs calculées par un programme. Les deux notions sont donc utilisées afin d'analyser les programmes fonctionnels du premier ordre de manière statique. Cependant, elles diffèrent pour deux principales raisons. La première étant qu'une quasi-interprétation est définie pour tout symbole d'un programme donné. La deuxième repose sur la propriété sous-terme introduite en I.1.6 dont sont dotées les quasi-interprétations. En effet, les

sup-interprétations sont monotones mais rien ne les oblige à être sous terme. En particulier, la sup-interprétation $\theta$ du symbole de fonction `minus` de l'exemple 1 pourra être choisie comme étant égale à $\theta(\texttt{minus})(X, Y) = X$, puisque la fonction calculée par `minus` retourne en sortie une valeur dont la taille n'excède jamais celle de son premier argument d'entrée, tandis que la quasi-interprétation $(\!|\texttt{minus}|\!)(X, Y)$ sera, dans le meilleur des cas, choisie comme étant égale à $\max(X, Y)$. Une telle différence de granularité engendre une intensionnalité plus forte des sup-interprétations par rapport aux quasi-interprétations. Intuitivement, plus la borne supérieure fournie par nos outils tend vers la taille exacte des valeurs calculées, plus le nombre d'algorithmes capturés par des méthodes formelles utilisant l'outil en question augmente.

**Théorème II.20.** *Toute quasi-interprétation additive est une sup-interprétation.*

*Démonstration.* La proposition I.21, implique que la condition 3 de la définition II.5 est respectée tandis que l'additivité garantit la validité du lemme II.4 et, par conséquent, permet de vérifier la condition 2 de la définition II.5. La définition I.13 des quasi-interprétations impose que l'assignation correspondante soit monotone. Par conséquent, la condition 1 de la définition II.5 est, elle aussi, vérifiée. □

Une conséquence particulièrement intéressante de ce théorème concerne le problème de la synthèse, introduit en section I.3 pour les quasi-interprétations, et qui consiste à trouver une sup-interprétation pour un programme donné. Ce problème est crucial à partir du moment où nous cherchons à automatiser l'analyse de la complexité des programmes. Comme nous l'avons démontré dans la section I.3, la synthèse de quasi-interprétations est décidable pour peu que l'on considère des polynômes à degré borné sur les réels. Puisque toute quasi-interprétation additive est une sup-interprétation, on obtient des heuristiques pour la synthèse de sup-interprétations en utilisant des algorithmes permettant de synthétiser des quasi-interprétations, comme celui décrit dans [Ama05]. La synthèse de sup-interprétations est un problème très difficile, indécidable en général, puisque connaître une sup-interprétation pour tout programme revient à déterminer si le programme termine où pas. Une étude de la synthèse des quasi-interprétations passe donc obligatoirement par une restriction à certaines classes de fonctions, comme dans les résultats de la section I.3 sur la synthèse de quasi-interprétations. Enfin, il est intéressant de constater que l'étude de la synthèse de sup-interprétations pour des programmes ne respectant pas la propriété de sous-terme reste, pour l'instant, un problème encore ouvert qui sera traité en section II.5.

**Théorème II.21.** *Tout programme admettant une quasi-interprétation polynomiale est quasi-amical.*

*Démonstration.* Le théorème précédent établit que toute quasi-interprétation définit une sup-interprétation. Il reste à constater que toute quasi-interprétation définit un poids, puisqu'une quasi-interprétation est sous-terme et monotone. □

**Proposition II.22.** *Il existe des programmes quasi-amicaux n'admettant pas de quasi-interprétation polynomiale.*

*Démonstration.* Le programme de division de l'exemple 1 est quasi-amical mais n'admet pas de quasi-interprétation. En effet, supposons qu'il admette une quasi-interprétation additive $(\!|-|\!)$. La dernière définition doit vérifier :

$$\begin{aligned}
(\!|\mathtt{q}(\mathbf{S}(u),\mathbf{S}(v))|\!)^* &= (\!|\mathtt{q}|\!)(U+k,V+k) && \text{avec } k \text{ constante}\\
&\geq (\!|\mathbf{S}(\mathtt{q}(\mathtt{minus}(u,v),\mathbf{S}(v)))|\!)^* && \text{Par Dfn de } (\!|-|\!)\\
&\geq k + (\!|\mathtt{q}|\!)(\max(U,V),V+k) && \text{Propriété sous-terme}\\
&> (\!|\mathtt{q}|\!)(U+k,V+k) && \text{En prenant } V \geq U+1
\end{aligned}$$

Par conséquent, nous obtenons une contradiction et $\mathtt{q}$ n'admet pas de quasi-interprétation. □

De nombreux autres algorithmes « naturels » calculant des fonctions polynomiales n'admettent pas de quasi-interprétation. C'est le cas des algorithmes calculant le plus grand commun diviseur et le Quicksort décrits dans les exemples 20 et 29. Nous donnons ici un nouvel exemple d'un tel programme :

**Exemple 30** (Logarithme)**.**

$$\begin{aligned}
&\mathtt{half}(x) = \textbf{Case } x \textbf{ of}\\
&\qquad \mathbf{0} \to \mathbf{0}\\
&\qquad \mathbf{S}(\mathbf{0}) \to \mathbf{0}\\
&\qquad \mathbf{S}(\mathbf{S}(y)) \to \mathbf{S}(\mathtt{half}(y))\\
&\mathtt{log}(x) = \textbf{Case } x \textbf{ of}\\
&\qquad \mathbf{0} \to \mathbf{0}\\
&\qquad \mathbf{S}(\mathbf{0}) \to \mathbf{0}\\
&\qquad \mathbf{S}(\mathbf{S}(y)) \to \mathbf{S}(\log(\mathtt{half}(\mathbf{S}(\mathbf{S}(y)))))
\end{aligned}$$

*Si le programme admet une quasi-interprétation additive $(\!|-|\!)$ alors elle doit vérifier pour*

*la dernière règle :*

$$\begin{aligned}(\!|\mathtt{log}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= (\!|\mathtt{log}|\!)(2 \times k + Y) \\ &\geq k + (\!|\mathtt{log}|\!)((\!|\mathtt{half}|\!)(2 \times k + Y)) \\ &\geq k + (\!|\mathtt{log}|\!)(2 \times k + Y)\end{aligned}$$

*avec* $(\!|\mathbf{S}|\!)(X) = X + k$ *et* $k \geq 1$. *Le programme n'admet donc pas de quasi-interprétation additive.*

Le théorème II.21 a une autre conséquence importante. Toutes les caractérisations valables dans la section I.2 le sont aussi dès lors que l'on considère des sup-interprétations quasi-amicales. On obtient donc de nouvelles caractérisations de la classe des fonctions calculables en temps polynomial et de la classe des fonctions calculables en espace polynomial à l'aide des sup-interprétations. Parmi ces caractérisations, citons plus particulièrement :

**Théorème II.23.**

- *L'ensemble des fonctions calculées par un programme quasi-amical admettant une sup-interprétation et ordonné par* $\prec_{rpo}$ *où chaque symbole de fonction a un statut produit est exactement l'ensemble des fonctions calculables en temps polynomial.*
- *L'ensemble des fonctions calculées par un programme quasi-amical admettant une sup-interprétation et ordonné par* $\prec_{rpo}$ *est exactement l'ensemble des fonctions calculables en espace polynomial.*

*Démonstration.* Ce résultat est une conséquence directe des théorèmes II.10 et II.20. □

### II.3.2 Application aux paires de dépendance

Nous allons dès à présent définir la notion de paire de dépendance utilisée dans [AG00] afin de prouver la terminaison des programmes de manière automatique. Cette notion permet de définir un critère de terminaison général intensionnellement très puissant. Ce critère est complet, c'est-à-dire qu'il permet de démontrer la terminaison de tous les programmes qui terminent, et il est, par conséquent, indécidable. Il est donc nécessaire de spécifier des applications d'un tel critère. Une application intéressante réside dans la combinaison de ce critère de terminaison avec la notion de sup-interprétation. Nous obtenons ainsi un critère efficace qui nous permet, de surcroît, d'obtenir une nouvelle caractérisation de la classe des fonctions calculables en espace polynomial.

**Définition II.24.** *Étant donné un programme* $\boldsymbol{p}$, *une paire de dépendance est un couple*

$$\langle \mathtt{f}(p_1, \cdots, p_n), \mathtt{g}(e_1, \cdots, e_m) \rangle$$

*tel que* $\mathtt{g}(e_1, \cdots, e_m)$ *est activé par* $\mathtt{f}(p_1, \cdots, p_n)$*. On définit le graphe des paires de dépendance par :*

- *Les noeuds sont les paires de dépendance.*
- *Soient* $u = \langle \mathtt{f}_1(p_1, \cdots, p_n), \mathtt{f}_2(e_1, \cdots, e_m) \rangle$ *et* $v = \langle \mathtt{f}_3(q_1, \cdots, q_k), \mathtt{f}_4(d_1, \cdots, d_l) \rangle$*, deux paires de dépendance, une arête relie* $u$ *à* $v$ *si* $\mathtt{f}_2 = \mathtt{f}_3$*.*

*Un cycle de paires de dépendance est défini comme étant un cycle du graphe des paires de dépendance. Par la suite, on dira qu'une paire de dépendance* $u$ *est impliquée dans un cycle si* $u$ *appartient à ce cycle dans le graphe des paires de dépendance.*

**Remarque 5.** *Une fraternité* $\mathsf{C}[\mathtt{f}_1(\overline{e_1}), \ldots, \mathtt{f}_n(\overline{e_n})]$ *activée par* $\mathtt{f}(p_1, \cdots, p_n)$ *correspond à* $n$ *paires de dépendance* $\langle \mathtt{f}(p_1, \cdots, p_n), \mathtt{f}_i(\overline{e_i}) \rangle$ *qui sont toutes impliquées dans au moins un cycle du graphe des paires de dependance.*

Nous rappelons ici quelques notions de base sur les quasi-ordres :

**Définition II.25** (Quasi-ordre)**.** *Étant donnée une relation binaire* $\geq$ *sur les expressions,* $\geq$ *est un quasi-ordre si elle vérifie les propriétés suivantes :*

- $\forall x,\ x \geq x$ *(Réflexivité)*
- $\forall x, y, z,\ (x \geq y \wedge y \geq z) \Rightarrow (x \geq z)$ *(Transitivité)*

*À tout quasi-ordre* $\geq$*, on associe une relation d'ordre stricte* $>$ *définie comme étant sa restriction aux couples d'éléments distincts. En d'autres termes, la relation* $>$ *est transitive et irréflexive.*

**Définition II.26** (Quasi-ordre monotone)**.** *Étant donné un quasi-ordre* $\geq$ *sur les expressions,* $\geq$ *est un quasi-ordre monotone s'il vérifie la propriété suivante pour tout symbole* $b$ *d'arité* $n$ *:*

$$\forall i \in \{1, \ldots, n\}\,,\ e_i \geq d_i \Rightarrow b(e_1, \cdots, e_n) \geq b(d_1, \cdots, d_n)$$

**Définition II.27** (Quasi-ordre clos par substitution)**.** *Étant donné un quasi-ordre* $\geq$ *sur les expressions,* $\geq$ *est clos par substitution s'il vérifie la propriété suivante pour toute substitution* $\sigma$ *:*

$$e \geq d \Rightarrow e\sigma \geq d\sigma$$

**Définition II.28** (Quasi-ordre bien-fondé)**.** *Étant donnés un quasi-ordre* $\geq$ *sur les expressions et sa relation stricte* $>$*,* $\geq$ *est bien-fondé (ou noethérien) s'il n'existe pas de suite infinie* $(x_n)_{n \in \mathbb{N}}$ *d'expressions vérifiant* $\forall n \in \mathbb{N},\ x_n > x_{n+1}$

Le théorème suivant est dû à Arts et Giesl [AG00] :

**Théorème II.29.** *Un programme* $\boldsymbol{p}$ *termine s'il existe un quasi-ordre bien-fondé monotone et clos par substitution* $\geq_{q.o.}$ *tel que :*

1. *Pour toute expression maximale* $e$ *activée par* $\mathtt{f}(\overline{p})$, $\mathtt{f}(\overline{p}) \geq_{q.o.} e$
2. *Pour toute paire de dépendance* $\langle s, t\rangle$, $s \geq_{q.o.} t$
3. *Pour tout cycle dans le graphe des paires de dépendance, il existe une paire de dépendance* $\langle s, t\rangle$ *vérifiant* $s >_{q.o.} t$

De nombreux outils démontrant la terminaison des programmes, dont entre autres AProVE [GSKT06], CiME [CMMU03] et TTT [HM04], utilisent des applications de cette méthode. Car, le critère est intrinsèquement indécidable ; il suffit de s'en convaincre en prenant le quasi-ordre $\geq_{q.o.}$ comme étant la relation de réécriture induite par les règles d'évaluation du programme. Ainsi, la méthode est complète puisqu'elle permet de démontrer la terminaison de n'importe quel programme. L'indécidabilité de la terminaison implique donc l'indécidabilité du critère. Il est donc nécessaire de spécialiser le quasi-ordre de manière à obtenir une application des paires de dépendance et une spécification du théorème présenté ci-dessus. Dans ce qui suit, nous allons utiliser les sup-interprétations et le critère quasi-amical afin de dériver une telle application.

**Définition II.30** (Appels récursifs strictement bornés)**.** *Un programme* $\boldsymbol{p}$ *a des appels récursifs strictement bornés s'il admet une sup-interprétation* $\theta$ *et un poids* $\omega$ *dans* ***Max-Poly***$\{\mathbb{N}\}$ *tels que :*
- $\boldsymbol{p}$ *a des appels récursifs bornés,*
- *Pour tout cycle du graphe des paires de dépendance, il existe une paire de dépendance de la forme* $\langle \mathtt{f}(p_1, \cdots, p_n), \mathtt{g}(e_1, \cdots, e_m)\rangle$ *telle que*

$$\omega_{\mathtt{f}}(\theta^*(p_1), \cdots, \theta^*(p_n)) > \omega_{\mathtt{g}}(\theta^*(e_1), \cdots, \theta^*(e_m))$$

**Théorème II.31.** *Un programme qui a des appels récursifs strictement bornés termine.*

*Démonstration.* Remarquons d'abord que le lemme II.13 est toujours valable lorsque l'on considère un programme avec des appels récursifs strictement bornés ; la seule différence résidant dans l'inégalité stricte. En appliquant le lemme II.13, pour deux états successifs $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ et $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$ de l'arbre des appels impliquant le même symbole de fonction, on obtient :

$$\omega_{\mathtt{f}}(\theta^*(u_1), \cdots, \theta^*(u_n) > \omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n))$$

Comme les assignations considérées sont dans **Max-poly** $\{\mathbb{N}\}$, la condition sur les appels strictement bornés implique que tout cycle de paires de dépendance décroit le poids d'au moins 1. En conséquence, il n'y a pas de boucle infinie durant l'exécution du programme puisque le nombre d'occurrences de chaque cycle commençant par $\mathtt{f}(p_1, \cdots, p_n)\sigma$, pour

une substitution $\sigma$ donnée, est borné par $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma))$. Donc le programme termine. □

**Remarque 6.** *Ce théorème peut aussi être appliqué à des sup-interprétations non polynomiales. Dans un tel cas de figure, il suffit de considérer des fonctions sur les entiers naturels afin de préserver les propriétés de bonne-fondaison.*

Pour terminer cette sous-section, nous allons étudier la combinaison de la méthode des paires de dépendance avec des sup-interprétations polynomiales sur les entiers naturels, obtenant ainsi une nouvelle caractérisation des fonctions calculables en espace polynomial.

**Lemme II.32.** *Étant donné un programme quasi-amical à appels récursifs strictement bornés, la taille de chaque branche de l'arbre des appels est bornée polynomialement par la taille des entrées, où la taille d'une branche est définie comme étant la somme des tailles de tous ses états et où la taille des entrées est définie comme étant la taille des valeurs contenues dans la racine de l'arbre des appels.*

*Démonstration.* Le théorème II.14 implique que la taille de toute valeur d'un état de l'arbre des appels est bornée polynomialement par la taille des entrées. En d'autres termes, il existe un polynôme $R$ tel que pour tout état $\langle \mathtt{g}, v_1, \cdots, v_k\rangle$ de l'arbre des appels de racine $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$, on ait :

$$\forall i \in \{1, \ldots, k\}\,,\ |v_i| \leq R(\max_{j=1..n}(|u_j|))$$

Il en résulte que la taille de tout état est borné par $Q(\max_{j=1..n}|u_j|)$ avec $Q(X) = m \times R(X)$ et $m$ l'arité maximale d'un symbole du programme. Dans la preuve du théorème II.31, nous avons démontré que tout cycle, dont le premier état est $\langle \mathtt{g}, v_1, \cdots, v_k\rangle$, possède au plus $\omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_k))$ occurrences (ce nombre étant borné par $\omega_{\mathtt{g}}(\alpha \times |v_1|, \ldots, \alpha \times |v_k|)$ d'après le lemme II.4). Il résulte de ceci que tout cycle dont le premier état est $\langle \mathtt{g}, v_1, \cdots, v_k\rangle$ a au plus $\omega_{\mathtt{g}}(\alpha \times Q(\max_{j=1..n}|u_j|), \ldots, \alpha \times Q(\max_{j=1..n}|u_j|))$ occurrences. À présent, posons $\omega(X) = \max_{\mathtt{g}\in Fct}(\omega_{\mathtt{g}}(\alpha \times Q(X), \ldots, \alpha \times Q(X)))$ lorsque $\omega_{\mathtt{g}}$ est explicitement défini pour un symbole de fonction $\mathtt{g}$ donné. Soit $A$ le nombre maximal de cycles dans le programme (le nombre $A$ est considéré comme étant une constante puisqu'il dépend uniquement de la taille du programme). Nous savons que le nombre d'états dans une branche de premier état $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ est borné par $A \times \omega(\max_{j=1..n}(|u_j|))$. Pour conclure, $A \times \omega(\max_{j=1..n}(|u_j|)) \times Q(\max_{j=1..n}(|u_j|))$ est le polynôme requis bornant la taille de chaque branche de l'arbre des appels. □

**Théorème II.33.** *L'ensemble des fonctions calculées par des programmes quasi-amicaux à appels récursifs strictement bornés est exactement l'ensemble des fonctions calculables en espace polynomial.*

*Démonstration.* D'après le lemme II.32, nous savons que la taille de chaque branche et de chaque état de l'arbre des appels est bornée polynomialement par la taille des entrées. L'évaluation du programme correspond à un parcours en profondeur de l'arbre des appels. Il résulte de ceci que l'ensemble des fonctions calculées par des programmes quasi-amicaux à appels récursifs strictement bornés est inclus dans Pspace. La preuve de complétude est inspirée par la preuve de la caractérisation contenue dans [BMM07] et présentée dans le théorème I.36 de la section I.2.5. Elle utilise des machines à registres parallèles (Parallel Register Machines ou PRM). Savitch [Sav70] et Chandra, Kozen et Stockmeyer [CKS81] ont démontré que l'ensemble des fonctions calculées par une PRM en temps polynomial correspond exactement à l'ensemble des fonctions calculable par une Machine de Turing en espace polynomial. Nous laissons le soin au lecteur de vérifier que le programme donné dans la preuve du théorème I.36 et qui simule les PRM par un programme de notre langage est bien quasi-amical à appels récursifs strictement bornés. □

**Remarque 7.** *Cette caractérisation est intensionnellement plus forte que la caractérisation donnée dans le théorème I.36. En effet, de nombreux programmes naturels ne terminant pas l'extension lexicographique d'un ordre récursif sur les chemins (RPO) possèdent sont quasi-amicaux à appels récursifs strictement bornés. En particulier, le programme calculant le plus grand commun diviseur de l'exemple 20 ne termine par RPO mais on peut démontrer qu'il est à appel récursifs strictement bornés en prenant la sup-interprétation suivante* $\theta(\mathbf{0}) = 0$*,* $\theta(\mathbf{S})(X) = X + 1$*,* $\theta(\mathtt{minus})(X, Y) = X$ *et en choisissant le poids* $\omega_{\mathtt{gcd}}(X, Y) = X + Y$*.*

### II.3.3 Application au size-change principle

Le size-change principle (ou SCP) introduit par Jones et al. [LJBA01] est un critère permettant de montrer la terminaison de programmes fonctionnels en vérifiant qu'un programme donné ne possède pas de flot d'exécution infini. Puisque la condition sur les appels récursifs strictement bornés essaie de contrôler ensemble les arguments d'un appel récursif, elle est, en un sens, plus proche de la méthode des paires de dépendance que de celle du SCP. Cette méthode a été adaptée dans des travaux plus récents [AK03, Ave06] à des programmes impératifs. Pour une comparaison détaillée entre le SCP et les paires de dépendance, nous invitons le lecteur à consulter la référence suivante [TG03].

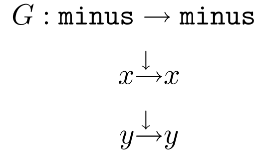

Fig. II.4: SCP graphe du programme minus

Adoptant la même philosophie que dans la précédente section, nous allons appliquer les sup-interprétations au SCP afin de prouver la terminaison d'un plus grand nombre d'algorithmes.

**Définition II.34** (SCP graphes et multi-chemins)**.** *Étant donnés $>_{w.f.o.}$ un quasi-ordre bien-fondé et sa clôture réflexive $\geq_{w.f.o.}$, un programme* **p** *et deux symboles de fonction* f *et* g *de* **p***, d'arités respectives $n$ et $m$, tels que l'expression* $\mathtt{g}(d_1, \cdots, d_m)$ *est activée par* $\mathtt{f}(p_1, \cdots, p_n)$*, les $d_1, \cdots, d_m$ représentant des expressions et les $p_1, \cdots, p_n$ des motifs, un SCP graphe de* f *vers* g *est un graphe bipartite, noté $G : \mathtt{f} \to \mathtt{g}$, des arguments $x_1, \cdots, x_n$ de* f *vers les arguments $y_1, \cdots, y_m$ de* g *où :*

- *Les noeuds sont les arguments $x_1, \cdots, x_n, y_1, \cdots, y_m$.*
- *Il y a une arête de $x_i$ vers $y_j$ si et seulement si $p_i \geq_{w.f.o.} d_j$.*
- *De plus, si $p_i >_{w.f.o.} d_j$, alors l'arête est étiquetée par $\downarrow$.*

*Un SCP multi-chemin est une séquence potentiellement infinie $G_1, G_2 \ldots$ de SCP graphes vérifiant pour tout $i$, $G_i$ est un SCP graphe de $\mathtt{f}_i$ vers $\mathtt{f}_{i+1}$ et $G_{i+1}$ est un SCP graphe de $\mathtt{f}_{i+1}$ vers $\mathtt{f}_{i+2}$. Intuitivement, un multi-chemin correspond à des appels successifs de fonctions pouvant advenir lors de l'exécution d'un programme. Un thread d'un SCP multi-chemin est défini comme étant une chemin d'arêtes connectées les unes aux autres dans le SCP multi-chemin.*

Constatons d'abord que, pour un programme donné, il existe seulement un nombre fini de SCP graphes.

**Exemple 31.** *Définisson $>_{w.f.o.}$ comme étant une relation d'ordre stricte bien-fondée sur la taille des valeurs, i.e. $u >_{w.f.o.} v$ si et seulement si $u, v \in \mathcal{V}$ et $|u| > |v|$, alors le symbole de fonction* minus *de l'exemple 1 possède un seul SCP graphe défini par : $G, \ldots, G$ est un SCP multi-chemin. $x \xrightarrow{\downarrow} x, \ldots,\ x \xrightarrow{\downarrow} x$ est un thread de ce multi-chemin.*

**Théorème II.35** ( [LJBA01]). *Un programme **p** termine si tout SCP multi-chemin infini possède un thread avec un nombre infini d'arêtes étiquetées par* $\downarrow$.

À présent, nous allons combiner ce théorème et la notion de sup-interprétation.

**Définition II.36** ($\theta$ SCP graphes). *Étant donnés un programme **p** et une sup-interprétation $\theta$, un $\theta$-SCP graphe, noté $G_\theta : \mathtt{f} \to \mathtt{g}$, est un SCP graphe $G : \mathtt{f} \to \mathtt{g}$ dont l'ordre bien-fondé sur les expressions $\geq_{w.f.o.}$ est défini par :*

- *$e \geq_{w.f.o.} d$ si et seulement si $\theta^*(e) \geq \theta^*(d)$*
- *$e >_{w.f.o.} d$ si et seulement si $\theta^*(e) > \theta^*(d)$*

*Un $\theta$-SCP multi-chemin est une séquence potentiellement infinie $G^1_\theta, G^2_\theta, \ldots$ de $\theta$-SCP graphes.*

**Théorème II.37.** *Étant donnée une sup-interprétation $\theta$ dont le codomaine est inclus dans l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, un programme **p** termine si tout $\theta$-SCP multi-chemin infini possède un thread avec une infinité d'arêtes étiquetées par $\downarrow$.*

*Démonstration.* La propriété de bonne-fondaison considérée dans le théorème II.35 est remplacée par le fait qu'une sup-interprétation d'une expression close est un entier naturel. Ainsi, une arête $\theta^*(p_i)\xrightarrow{\downarrow}\theta^*(e_j)$ du $\theta$-SCP graphe $G^k_\theta$ de $\mathtt{f}_k$ vers $\mathtt{f}_{k+1}$ correspond à l'activation d'une expression $\mathtt{f}_{k+1}(e_1, \cdots, e_m)$ par $\mathtt{f}_k(p_1, \cdots, p_n)$. Par définition de $\downarrow$, $\theta^*(p_i)\xrightarrow{\downarrow}\theta^*(e_j)$ si et seulement si $\theta^*(p_i) > \theta^*(e_j)$. Comme nous travaillons sur l'ensemble des entiers naturels, cette inégalité stricte correspond à une décroissance d'une constante fixée. Par hypothèse, tout multi-chemin infini possède au moins un thread avec une infinité d'arêtes de ce type et, par conséquent, le programme termine. □

Ce théorème s'inspire du size-change principle. Cependant, il ne correspond pas uniquement à une instance du théorème II.35. En effet, Jones et al. considèrent des ordres bien-fondés sur les valeurs tandis que le théorème II.37 permet de traiter une expression quelconque pour peu que l'on connaisse sa sup-interprétation. Ainsi, notre critère démontre la terminaison d'un plus grand nombre d'algorithmes et d'augmenter fortement l'intensionnalité du size-change principle comme le démontre l'exemple qui suit :

**Exemple 32.** *En choisissant une sup-interprétation $\theta$ vérifiant $\theta(\mathtt{minus})(X,Y) = X$ et $\theta(\mathbf{S})(X) = X + 1$, le programme $\mathtt{q}$ de l'exemple 1 possède trois $\theta$-SCP graphes définis par : Les $\theta$-SCP multi-chemins infinis commençant par un appel du symbole de fonction $\mathtt{q}$ sont tous de la forme $G^{3*}_\theta, G^{2*}_\theta$, $G^{1*}_\theta$ où $G^*$ est la notation utilisée pour représenter un nombre potentiellement infini d'occurrences de $G$. Cependant, ils contiennent tous un thread de la forme $(u\xrightarrow{\downarrow}u)^*$,$u\xrightarrow{\downarrow}x$,$(x\xrightarrow{\downarrow}x)^*$ avec un nombre infini d'arêtes étiquetées par $\downarrow$. La méthode initiale du size-change principle ne fonctionne pas sur ce programme*

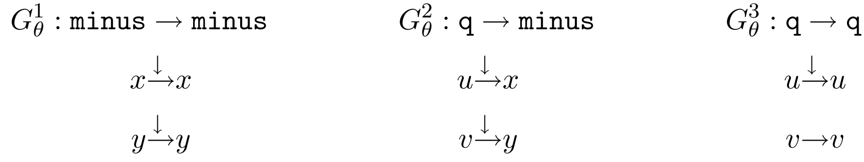


FIG. II.5: $\theta$-SCP graphe du programme q

*car seules les valeurs peuvent être comparées. On ne peut donc pas montrer que l'appel récursif du symbole de fonction* q*, de la forme* $\mathbf{S}(\mathtt{q}(\mathtt{minus}(z, u), \mathbf{S}(u)))$*, décroit puisque l'on ne peut pas comparer le symbole de fonction* minus *avec des valeurs.*

## II.4 Une caractérisation des classes de complexité NC$^k$

### II.4.1 Rappels sur ALogTime, NC$^k$ et NC

Dans cette section, nous utiliserons la notation $\log(n)$ pour désigner $\lceil \log_2(n+1) \rceil$, la notation $\lfloor x \rfloor$ pour représenter le plus grand entier inférieur ou égal à $x$ et la notation $\lceil x \rceil$ pour représenter le plus petit entier supérieur ou égal à $x$.

Nous présentons la notion de Machine de Turing Alternante à accès aléatoire (*Random Access Alternating Turing Machine* ou ATM) introduite par Chandra et al. [CKS81] et étudiée par [Ruz81]. Les ATM sont le modèle de calcul de base utilisé pour caractériser les petites classes de complexité parallèles. Une ATM est une Machine de Turing particulière possédant des rubans d'entrée, dont l'accès est aléatoire et en lecture seule, ainsi que des rubans de travail. Aucun ruban de sortie n'est spécifié car la machine rend un seul bit en sortie. L'accès aléatoire signifie que l'on peut accéder directement à une valeur contenue sur le ruban d'entrée en spécifiant son adresse. On distingue les états d'une ATM en trois catégories principales : les états conjonctifs, les états disjonctifs et les états de lecture. Une configuration d'une ATM à $n$ rubans est un $n+1$-uplet de la forme $(q, \epsilon_1, \ldots, \epsilon_n)$ où $q$ représente l'état de la machine et $\epsilon_i \in \{0,1\}^*$ représente le contenu du i-ème ruban.

Le calcul d'une ATM s'effectue en deux étapes successives. La première correspond au déploiement d'un arbre $T$ de configurations indépendant des entrées. La deuxième étape consiste à évaluer le calcul en utilisant l'arbre des configurations $T$ par rapport aux entrées. Étant donnée une fonction $F : \{0,1\}^* \to \{0,1\}^*$, $F_{bit}$ est définie comme la fonction qui au couple binaire $(x, u)$ associe $i$-ème bit de $F(x)$ si $u$ est la représentation binaire de l'entier $i$. La fonction $F$ est dite calculable bit par bit en temps $O(\log^k(n))$ et en espace $O(\log(n))$ si la fonction $F_{bit} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ est calculable par une

ATM en temps $O(\log^k(n))$ et en espace $O(\log(n))$, où $n$ est la taille de l'entrée. Dans le cas particulier où la constante $k$ est égale à 1, la fonction $F$ est dite calculable bit par bit dans *ALogTime*. D'après les travaux de Cook [Coo85], on dit qu'une fonction $\phi : \{0,1\}^* \to \{0,1\}^*$ est calculée par une ATM en temps $O(\log^k(n))$ et en espace $O(\log(n))$ (dans *ALogTime* si $k = 1$) si $\phi$ est calculable bit par bit en temps $O(\log^k(n))$ et en espace $O(\log(n))$ (bit par bit dans *ALogTime* si $k = 1$) et $\phi$ est bornée polynomialement, i.e. $|\phi(x)| = |x|^{O(1)}$ où la taille $|w|$ d'un mot $w$ est définie comme étant son nombre de bits.

À présent, nous allons introduire la notion de circuit. Un circuit $C_n$ est un graphe orienté acyclique construit à partir de portes (ou noeuds) d'entrée et de sortie ainsi que de portes booléennes *0*, *1*, *And*, *Or* et *Not*. Chaque porte booléenne a un degré d'entrée inférieur ou égal à deux et un degré de sortie égal à un. Les portes d'entrées ont un degré d'entrée égal à zéro et les portes de sortie ont un degré de sortie égal à zéro. Nous considèrerons des circuits a $n$ portes d'entrée et à $g(n)$ portes de sortie où $g$ est une fonction satisfaisant $g(n) = O(n^c)$ avec $c$ une constant supérieure ou égale à 1. Un circuit $C_n$ calcule une fonction $f_n : \{0,1\}^n \to \{0,1\}^{g(n)}$. Nous cherchons à considérer des fonctions de $\{0,1\}^* \to \{0,1\}^*$ et, par conséquent, il nous faut définir la notion de famille de circuits. Une famille de circuits est une suite de circuits $C = (C_n)_{n\in\mathbb{N}}$, calculant une famille de fonctions finies $(f_n)_{n\in\mathbb{N}}$ à valeur sur $\{0,1\}^*$. On dit qu'une fonction $f$ est calculée par une famille de circuits $(C_n)_{n\in\mathbb{N}}$ si la restriction de $f$ à des entrées de taille $n$ est calculée par $C_n$. Autrement dit, $f$ est calculée par la famille de circuits $(C_n)_{n\in\mathbb{N}}$ calculant les fonctions finies $(f_n)_{n\in\mathbb{N}}$ si pour tout mot $w \in \{0,1\}^*$, $f(w) = f_{|w|}(w)$. La complexité d'un circuit dépend de sa profondeur, qui correspond au plus long chemin reliant une porte d'entrée à une porte de sortie, et de sa taille, le nombre de portes le constituant. Cependant, la notion de famille de circuit doit être combinée avec une notion d'uniformité. L'uniformité est une restriction de la notion de famille de circuit à une notion cohérente ne permettant pas d'effectuer des calculs indécidables comme le montre la preuve de la proposition ci-dessous :

**Proposition II.38** ([Pap94]). *Il existe des langages indécidables reconnus par une famille de circuits de taille polynomiale.*

*Démonstration.* Soit $L$ un langage indécidable de $\{0,1\}^*$. On définit le langage $L'$ comme étant la langage contenant des mots binaires de la forme $1^n$ lorsque l'écriture binaire de l'entier $n$ appartient à $L$. Il résulte de cette construction que $L'$ est aussi indécidable. $L'$ est reconnu par la famille polynomiale de circuits $(C_n)_{n\in\mathbb{N}}$ définie par :

- Si $1^n \in L'$ alors on définit $C_n$ sur les entrées $x_1, \cdots, x_n$ et la sortie $y$ par $y = x_1$ *And* $(x_2$ *And* $(\dots$ *And* $x_n))$

– Sinon $C_n$ est défini sur les entrées $x_1, \cdots, x_n$ et la sortie $y$ par $y = 0$.

Cette famille de circuits de taille polynomiale reconnait bien le langage $L'$ qui est indécidable. □

Dans cette optique différentes restrictions appelées conditions d'uniformité ont été introduites. Une condition d'uniformité garantit l'existence d'une procédure qui, étant donné le nombre $n$, peut produire une description du circuit $C_n$.

Une des premières conditions d'uniformité introduite, appelée log-space uniformité, fut de supposer, qu'étant donnée une entrée de taille $n$, une Machine de Turing peut générer le circuit $C_n$ correspondant en espace logarithmique en $n$. D'autres critères d'uniformité ont par la suite été établis comme l'$U_B$-uniformité [Bor77] ou l'$U_{BC}$-uniformité [BC80]. D'autres conditions d'uniformité ont aussi été étudiées dans [BIS90]. Dans cette section, nous utiliserons la notion d'$U_{E^*}$-uniformité introduite par [Ruz81] et définie ci-dessous :

**Définition II.39.** *Étant donnée une famille de circuits $C = (C_n)_n$ de tailles et de profondeurs respectives $(|C_n|)_n$ et $(h(C_n))_n$, le langage étendu des connexions $L_{EC}$ est un ensemble de quadruplets $(n, g, p, y)$ où la porte indiquée par le chemin $p$ de taille $|p| \leq \log(|C_n|)$ en partant de la porte numérotée $g$ est de type $y$ dans le circuit $C_n$. $C$ est $U_{E^*}$-uniforme s'il existe une ATM reconnaissant $L_{EC}$ en temps $O(h(C_n))$ et en espace $O(\log(|C_n|))$.*

À présent, nous allons définir les classes de complexité parallèles les plus communément étudiées. La classe $NC$ (« Nick's Class ») est la classe des problèmes pouvant être résolus très rapidement avec un nombre raisonnable de processeurs ou, plus précisément, la classe des problèmes calculables en temps poly-logarithmique en la taille de l'entrée avec un nombre polynomial de processeurs. Formellement, $NC$ est définie comme étant la classe des fonctions calculables par une famille de circuits $U_{E^*}$-uniforme de taille bornée par $O(n^d)$, pour une constante $d$ donnée, et de profondeur bornée par $O(\log^j(n))$ pour une constante $j$ quelconque. La classe $NC$ est ensuite découpée en sous-classes, appelées $NC^k$ et définies comme étant les classes de fonctions calculables par une famille de circuits $U_{E^*}$-uniforme de taille bornée par $O(n^d)$, pour une constante $d$ donnée, et de profondeur bornée par $O(\log^k(n))$ pour $k$ fixé. La classe $NC$ peut aussi être définie comme étant l'union des classes $NC^k$, $NC = \cup_{k \in \mathbb{N}} NC^k$. La motivation première pour introduire de telles classes est la recherche de résultats de séparation. Pour [BIS90], $NC^1$ est la frontière où nous commençons à obtenir des résultats de séparation intéressants. $NC^1$ contient l'addition binaire, la soustraction, la somme préfixe d'opérateurs associatifs. Buss [Bus87] a démontré que l'évaluation de formules Booléennes est un problème complet pour $NC^1$. De nombreux autres problèmes naturels appartiennent aux différents niveaux de la hiérarchie $NC^k$. En particulier, le problème d'accessibilité dans un graphe

ou la recherche d'une forêt couvrante minimum d'un graphe sont des problèmes de $NC^2$. On obtient les inclusions de classes suivantes :

$$AC^0 \subseteq NC^1 \subseteq \mathrm{FL} \subseteq NC^2 \subseteq \ldots \subseteq NC^k \subseteq NC^{k+1} \subseteq \ldots \subseteq NC \subseteq \mathrm{FP}\textsc{time}$$

où FL est la classe des fonctions calculables par une Machine de Turing déterministe en espace logarithmique et $AC^0$ est la classe des fonctions calculables par un circuit de taille polynomiale, de profondeur constante avec des portes d'arité non-bornée. Les inclusions $AC^0 \subseteq NC^1$ et $NC^j \subseteq NC^{j+1}$ sont évidentes tandis qu'une preuve de $NC^1 \subseteq \mathrm{FL}$ est disponible dans [Coo85].

Comme le signale Cook dans [Coo85], la notion d'$U_{E^*}$-uniformité est la plus communément utilisée pour la raison qu'elle est la plus faible notion d'uniformité pour laquelle le théorème suivant a été démontré :

**Théorème II.40** (Ruzzo [Ruz81])**.** *Pour tout $k \geq 1$, une fonction $\phi : \{0,1\}^* \to \{0,1\}^*$ est calculable dans $NC^k$ si et seulement si $\phi$ est calculée par une ATM en temps $O(\log^k(n))$ et en espace $O(\log(n))$.*

En particulier :

**Théorème II.41.** *Une fonction $\phi : \{0,1\}^* \to \{0,1\}^*$ est dans $NC^1$ si et seulement si elle est calculée dans ALogTime.*

### Résultats obtenus

Désormais, nous allons établir une caractérisation des classes de complexité $NC^k$ en utilisant la notion de sup-interprétation. À cet effet, nous définissons une restriction sur les programmes considérés et nous démontrons que cette restriction permet de caractériser les fonctions calculables dans *ALogTime*, c'est-à-dire $NC^1$, ainsi que dans $NC^k$ pour tout $k \in \mathbb{N}$ et $NC$. La caractérisation de $NC^1$ a été établie dans [BMP06] et est, à notre connaissance, la première caractérisation d'une petite classe de complexité parallèle à l'aide d'un système de réécriture bien que l'on puisse trouver de nombreuses références bibliographiques fournissant des caractérisations de *ALogTime* à l'aide de modèles de calcul différents, comme les ATM ou encore les algèbres de fonctions. Compton et Laflamme [CL90] en ont donné une caractérisation basée sur des fonctions finies. Cependant, rares sont les caractérisations pour lesquelles l'analyse statique est envisageable. Bloch [Blo94] a donné une caractérisation de *ALogTime* en utilisant un schéma de récurrence ramifié de type « diviser pour régner ». Leivant et Marion [LM00] ont établi une caractérisation basée sur le principe de récurrence ramifiée qui est restreint une structure de données spécifique, les arbres binaires bien équilibrés. Clote [Clo95] a donné

une caractérisation des $NC^k$ basée sur un schéma de récurrence borné. Enfin, des caractérisations de $NC$ sont aussi fournies dans [Lei98, BKMO06]. Toutes ces caractérisations ont cependant un inconvénient majeur : Elles reposent sur des conditions syntaxiques peu expressives et possèdent donc une intensionnalité limitée ; peu d'algorithmes naturels sont capturés. Les caractérisations présentées dans cette section essaient d'apporter une réponse à ce problème.

### II.4.2 Programmes arborescents et sympathiques

#### Programmes arborescents

Nous commençons par définir un nouveau type d'assignation, la notion de poids faible. Lorsque l'on s'intéresse aux petites classes de complexité parallèle, les outils de notre étude se doivent de posséder une granularité très fine. De ce point de vue, la condition sous-terme sur les poids est beaucoup trop restrictive. Nous définissons donc le poids faible, un poids sans propriété sous-terme utilisé afin de contrôler le nombre d'appels récursifs successifs :

**Définition II.42.** *Un poids faible $\omega$ est une assignation partielle de domaine Fct, l'ensemble des symboles de fonction. À un symbole de fonction donné $\mathtt{f}$ d'arité $n$, elle assigne une fonction totale et monotone $\omega_{\mathtt{f}}$ de $(\mathbb{R}^+)^n$ dans $\mathbb{R}^+$.*

**Définition II.43.** *Un programme $\boldsymbol{p}$ est arborescent si et seulement s'il existe une sup-interprétation polynomiale et additive $\theta$, un poids faible polynomial $\omega$ et une constante $K > 1$ tels que pour toute fraternité $\mathsf{C}[\mathtt{g}_1(\overline{t_1}), \ldots, \mathtt{g}_r(\overline{t_r})]$ activée par $\mathtt{f}(p_1, \cdots, p_n)$ et pour toute substitution $\sigma$, les conditions suivantes sont satisfaites :*

- $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq 1$
- $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq K \times \omega_{\mathtt{g}_i}(\theta^*(\overline{t_i}\sigma)) \quad \forall 1 \leq i \leq r$
- *et il n'existe pas de symbole de fonction $\mathtt{h} \in \overline{e_i}$ tel que $\mathtt{h} \approx_{Fct} \mathtt{f}$*

*La constante $K$ est appelée coefficient arborescent de $\boldsymbol{p}$.*

**Remarque 8.** *Contrairement aux résultats obtenus pour des programmes quasi-amicaux, le fait de considérer des poids faibles, c'est-à-dire des poids sans la propriété sous-terme, ne garantit plus aucun contrôle sur la taille des valeurs calculées par un programme arborescent. En revanche, ce critère nous garantit des bornes logarithmiques sur le nombre d'appels récursifs successifs pouvant être appelés par le programme, comme nous le démontrerons dans le corollaire II.45.*

**Exemple 33.** *Nous proposons ici un exemple de programme arborescent calculant la somme préfixe. Un tel programme est un exemple typique de calcul pouvant être effectué*

*efficacement et de manière parallèle par des circuits. D'abord, supposons que* $\bigodot$ *est une opération associative binaire sur un ensemble A. La somme préfixe d'une liste* $[x_1, \ldots, x_n]$ *de n éléments de A, est notée* $x_1 \bigodot \ldots \bigodot x_n$*. Les listes à valeur dans l'ensemble A sont définies de manière classique par :*

$$\mathtt{List}(A) = [\,] \mid [A, \mathtt{List}(A)]$$

*Supposons aussi que nous disposions de deux opérateurs* **Left** *et* **Right***, dont le rôle est de diviser une liste en deux parties :*

$$[\![\mathbf{Left}]\!]([\,]) = [\,] \qquad [\![\mathbf{Left}]\!]([x_1, \ldots, x_n]) = [x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}]$$
$$[\![\mathbf{Right}]\!]([\,]) = [\,] \qquad [\![\mathbf{Right}]\!]([x_1, \ldots, x_n]) = [x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n]$$

*en utilisant la notion* $[x_1, \ldots, x_n]$ *en lieu et place de* $[x_1, [x_2, \ldots, x_n]]$*. La somme préfixe d'une liste se calcule de la manière suivante :*

$$\begin{aligned} \mathtt{sum}(z) = \mathbf{Case}\ z\ \mathbf{of}\ & [x] \to x \\ & [x, y, L] \to \mathtt{sum}(\mathbf{Left}([x, y, L])) \bigodot \mathtt{sum}(\mathbf{Right}([x, y, L])) \end{aligned}$$

*en utilisant une notation infixe pour l'opérateur* $\bigodot$*. Remarquons que le motif* $[x, y, L]$ *permet de capturer une liste de longueur au moins égale à deux.*

*Les symboles de constructeur et d'opérateur admettent la sup-interprétation suivante :*

$$\theta([\,]) = 0 \qquad \theta([X, L]) = X + L + 1$$
$$\theta(\mathbf{Left})(N) = \lfloor \frac{N}{2} \rfloor \qquad \theta(\mathbf{Right})(N) = \lceil \frac{N}{2} \rceil$$

*La taille d'une liste étant égale au nombre d'éléments qui la composent, on peut vérifier que pour toute liste L, on a* $|L| \leq \theta(L)$*. On pourrait aussi vérifier les inégalités suivantes* $|[\![\mathbf{Left}]\!](l)| \leq \theta(\mathbf{Left})(\theta^*(l))$ *et* $|[\![\mathbf{Right}]\!](l)| \leq \theta(\mathbf{Right})(\theta^*(l))$ *pour toute liste l. De tout ceci, il résulte que* `sum` *est un programme arborescent en prenant* $\omega_{\mathtt{sum}}(L) = L$ *et* $K = \frac{3}{2}$ *pour* $L \geq 2$*.*

Nous allons démontrer qu'un programme arborescent termine et possède une borne sur la longueur de ses dérivations au cours d'un appel récursif.

**Théorème II.44.** *Étant donné un programme arborescent* $\boldsymbol{p}$*, le programme* $\boldsymbol{p}$ *termine. En d'autres termes, pour tout symbole de fonction* `f` *et pour toute séquence de valeurs* $u_1, \cdots, u_n$ *appartenant à* $\mathcal{V}$*,* $[\![\mathtt{f}]\!](u_1, \cdots, u_n)$ *appartient à* $\mathcal{V}^*$*.*

*Démonstration.* Étant donné un programme arborescent **p**. Nous allons démontrer ce résultat par induction sur la précédence $\geq_{Fct}$. Soit $\mathtt{f}$ un symbole de fonction d'arité $n$, supposons par hypothèse d'induction que pour tout $\mathtt{g}$ tel que $\mathtt{f} >_{Fct} \mathtt{g}$ et pour toute séquence de valeur $\overline{u} \in \mathcal{V}$, on ait $[\![\mathtt{g}]\!](\overline{u}) \in \mathcal{V}^*$. À présent, nous allons considérer l'évaluation de $\mathtt{f}$ pour $n$ valeurs $v_1, \cdots, v_n \in \mathcal{V}$. Deux possibilités s'offrent à nous :

- Soit une règle du type $\mathtt{f}(x_1, \cdots, x_n) = \textbf{Case } x_1, \cdots, x_n \textbf{ of } p_1, \cdots, p_n \to e$ est appliquée, avec $\mathtt{f} >_{Fct} \mathtt{g}$ pour tout symbole de fonction $\mathtt{g}$ apparaissant dans $e$. Dans ce cas, il existe une substitution $\sigma$ telle que $\overline{p}\sigma = \overline{v}$ et en appliquant l'hypothèse d'induction sur la structure de $e$, on obtient que $[\![\mathtt{f}]\!](u_1, \cdots, u_n)$ appartient à $\mathcal{V}^*$.
- Soit on a une fraternité dans $e$, i.e. $e = \mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$, avec $\mathtt{g}_1 \approx_{Fct} \ldots \approx_{Fct} \mathtt{g}_r \approx_{Fct} \mathtt{f}$ et on vérifie d'après le critère arborescent II.43 les conditions suivantes :
  - $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq 1$
  - $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq K \times \omega_{\mathtt{g}_i}(\theta^*(\overline{t_i}\sigma)) \quad \forall 1 \leq i \leq r$

  La dernière condition combinée au lemme II.6 et à la monotonie des poids faibles, nous donne l'inégalité suivante :

$$\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq K \times \omega_{\mathtt{g}_i}(\theta^*([\![\overline{t_i}\sigma]\!]))$$

  Par conséquent, le $i$-ème appel en profondeur dans l'arbre des appels d'une fonction $\mathtt{g}$ équivalente à $\mathtt{f}$ sur des arguments $\overline{v}$, nous donne l'inégalité suivante :

$$\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq K^i \times \omega_{\mathtt{g}}(\theta^*(\overline{v}))$$

  La première condition, nous garantit un seuil d'arrêt pour le nombre de divisions possibles par la constante $K > 1$. Il résulte de ceci que l'on a au plus $\log_K(\omega_{\mathtt{f}}(\theta^*(\overline{u})))$ appels récursifs pour la branche en considération. Puisque le programme est sympathique, cette condition est valable pour toute branche d'appels récursifs (en effet, on vérifie que $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq K \times \omega_{\mathtt{g}_i}(\theta^*(\overline{t_i}\sigma))$ pour tout $i \leq r$) et, par conséquent, le programme termine.

□

**Corollaire II.45.** *Étant donné un programme arborescent **p** et un arbre des appels correspondant à une exécution de **p**, alors toute branche de l'arbre des appels, dont les états correspondent à des symboles de fonction équivalents pour la précédence $\geq_{Fct}$ et dont le premier état est $\langle \mathtt{g}, v_1, \cdots, v_k \rangle$, a une longueur (un nombre d'états) bornée par $\alpha \times \log(\omega_{\mathtt{g}}(\theta^*(\overline{v})))$ avec $\alpha$ constante.*

*Démonstration.* Dans la démonstration du théorème précédent, nous avons vu que la longueur d'une telle branche correspondant à des fonctions équivalentes du programmes

était bornée par $\log_K(\omega_{\mathtt{g}}(\theta^*(\overline{v})))$, si $\langle \mathtt{g}, v_1, \cdots, v_k\rangle$ est le premier état de cette branche. Il nous reste à utiliser l'identité $\log_2(x) = \frac{\log_K(x)}{\log_K(2)}$ et l'inégalité $\log_2(x) \leq \log(x)$ pour obtenir le résultat désiré. □

### Programmes sympathiques

Nous introduisons ici la notion de programme sympathique. Un programme sympathique est un programme dont les symboles de fonction et d'opérateur utilisés dans le contexte et dans les arguments des appels récursifs de toute fraternité possèdent une sup-interprétation bornée de manière affine par la taille des entrées. L'intuition derrière un tel critère est de borner la puissance de calcul des appels récursifs du programme.

**Définition II.46.** *Un programme* $\mathbf{p}$ *est sympathique si et seulement s'il existe une sup-interprétation additive et polynomiale* $\theta$ *telle que, pour toute fraternité de la forme* $\mathsf{C}[\mathtt{g}_1(\overline{t_1}), \ldots, \mathtt{g}_r(\overline{t_r})]$ *activée par* $\mathtt{f}(p_1, \cdots, p_n)$ *et pour tout symbole de fonction et d'opérateur* $b$ *apparaissant dans* $\mathsf{C}$ *ou dans les* $\overline{t_j}$*, il existe des constantes* $\alpha_i, \beta \in \mathbb{R}^+$ *vérifiant :*

$$\theta(b)(X_1, \cdots, X_n) \leq \sum_{i=1}^{n} \alpha_i \times X_i + \beta$$

**Remarque 9.** *L'intuition derrière un tel critère est de restreindre la puissance de calcul des programmes considérés. En effet, s'il borne logarithmiquement la profondeur des récurrences dans l'arbre des appels, le critère arborescent ne borne en aucune manière la taille des calculs effectués, comme l'illustre l'exemple suivant :*

$$\begin{aligned} \mathtt{f}(x, y) = \mathbf{Case}\ x, y\ \mathbf{of}\ & u, v \to \mathtt{f}(\mathtt{half}(u), \mathtt{sq}(v)) \\ & \mathbf{0}, v \to v \end{aligned}$$

*où* $[\![\mathtt{half}(\mathbf{S}^n(\mathbf{0}))]\!] = \mathbf{S}^{\lfloor n/2 \rfloor}(\mathbf{0})$ *et* $[\![\mathtt{sq}(\mathbf{S}^n(\mathbf{0}))]\!] = \mathbf{S}^{n^2}(\mathbf{0})$*. En prenant* $\theta(\mathtt{half})(X) = X/2$*,* $K = 2$ *et* $\omega_{\mathtt{f}}(X, Y) = X$*, on démontre que le programme est arborescent. Cependant, on a* $[\![\mathtt{f}(\mathbf{S}^n(\mathbf{0}), \mathbf{S}^m(\mathbf{0}))]\!] = \mathbf{S}^{(m^2)^{\log(n)}}(\mathbf{0}) = \mathbf{S}^{m^{2 \times \log(n)}}(\mathbf{0})$*. Or, nous cherchons à borner polynomialement la taille des valeurs calculées.*

La combinaison des propriétés arborescente et sympathique d'un programme implique que la taille des valeurs calculées est bornée polynomialement par la taille des entrées. Nous donnons l'intuition qui se cache derrière un tel résultat. Le critère sympathique impose que toutes les fonctions utilisées dans les appels récursifs soient bornées par une fonction affine. Le corollaire II.45 nous garantit qu'un programme arborescent possède au plus un nombre logarithmique d'appels récursifs. La composition d'un nombre logarithmique de fonction affines retourne une valeur dont la taille est au plus polynomiale.

**Lemme II.47.** *Étant donnée une sup-interprétation $\theta$ telle que le programme $\boldsymbol{p}$ soit arborescent et sympathique, il existe un polynôme $P$ tel que pour tout séquence de valeurs $u_1, \cdots, u_n$ et pour tout symbole de fonction* f *on ait :*

$$|[\![\texttt{f}]\!](u_1, \cdots, u_n)| \leq P(\max_{i=1..n}(|u_i|))$$

*Démonstration.* Étant donné un programme sympathique et arborescent et un symbole de fonction f. On suppose, par hypothèse d'induction qu'il existe un polynôme $P$ tel que pour tout symbole de fonction g d'arité $m$ satisfaisant $\texttt{f} >_{Fct} \texttt{g}$ et pour toute séquence de valeurs $u_1, \cdots, u_m$ on a $|[\![\texttt{g}]\!](u_1, \cdots, u_m)| \leq P(\max_{i=1..m}(|u_i|))$.

Supposons, sans perte de généralité, que f n'a pas de symbole équivalent autre que lui même. Définissons $R(X) = P^l(X)$ où $l$ est la profondeur maximale d'une expression $e$, pour toute expression $e$ activée par $\texttt{f}(\overline{p})$ n'étant pas une fraternité, et où $P^k$ dénote $k$ compositions du polynôme $P$. Intuitivement, $R$ est une borne supérieure sur les calculs qui quittent un appel récursifs de f. Si $\alpha$ est la constante du lemme II.4, $\alpha \times R$ est donc une « sup-interprétation partielle » de f, c'est-à-dire que $\alpha \times R$ fournit une sup-interprétation de f dans le cas où la définition appliquée ne correspond pas à un appel récursif. En effet, soit $v_1, \cdots, v_n$ une séquence de valeurs telle qu'il existe une substitution $\sigma$ et une expression $e$ activée par $\texttt{f}(\overline{p})$ n'étant pas une fraternité et vérifiant $\overline{p}\sigma = v_1, \cdots, v_n$ et $[\![e\sigma]\!] = u$, on observe que :

$$\begin{aligned}
\alpha \times R(\theta^*(\overline{v})) &\geq \alpha \times R(|\overline{v}|) && \text{Par monotonie de } R \\
&\geq \alpha \times |u| && \text{Par définition de } R \\
&\geq \theta^*(u) = \theta^*([\![e\sigma]\!]) && \text{D'après le lemme II.4}
\end{aligned}$$

Le programme étant sympathique, on peut montrer, par induction sur la structure d'une expression, que les arguments et les contextes ont des calculs bornés par une fonction de la forme $a \times m + b$, où $a$ et $b$ sont des constantes et $m$ est la taille des entrées. En appliquant le corollaire II.45, on sait que le nombre d'appels récursifs successifs de f est borné par $d \times \log(m)$, pour une constante $d$ fixée. Par conséquent, les fonctions calculées par la composition des contextes et des arguments d'une fraternité sont bornées par $Q(m) = m^{(d \times \log_2(a)+1)} + \beta \times (m^{(d \times \log_2(a))}) + \gamma$ avec $\beta$ et $\gamma$ des constantes ne dépendant que de $a$ et de $b$. Finalement, on définit $P'(X) = Q(\alpha \times R(\alpha \times Q(X))) + P(X)$, avec $\alpha$ la constante du lemme II.4.

Considérons la stratégie d'évaluation suivante :

$$\texttt{f}(u_1, \cdots, u_n) \xrightarrow{*} \texttt{C}[\texttt{f}(\overline{e_1}), \ldots, \texttt{f}(\overline{e_m})] \xrightarrow{*} [\![\texttt{f}]\!](u_1, \cdots, u_n)$$

où $\xrightarrow{*}$ est la relation d'évaluation induite par les définitions du programme et les $\mathtt{f}(\overline{e_j})$ sont des expressions dont l'évaluation n'appelle plus que des symboles de fonctions strictement plus petits que $\mathtt{f}$ pour la précédence $\geq_{Fct}$. Une telle réduction est possible puisque nous avons supposé qu'il n'existait aucun symbole équivalent à $\mathtt{f}$ et puisque les programmes considérés sont confluents et terminent d'après le théorème II.44.

Par construction, on obtient :

$$\begin{aligned}
|[\![\mathtt{f}]\!](u_1,\cdots,u_n)| &\leq \theta^*([\![\mathtt{f}]\!](u_1,\cdots,u_n)) && \text{Lemme II.7}\\
&\leq \theta^*(\mathsf{C}[\mathtt{f}(\overline{e_1}),\ldots,\mathtt{f}(\overline{e_m})]) && \text{Lemme II.6}\\
&= \theta^*(\mathsf{C})[\theta(\mathtt{f})(\theta^*(\overline{e_1})),\ldots,\theta(\mathtt{f})(\theta^*(\overline{e_m}))] && \text{Définition II.5}\\
&= \theta^*(\mathsf{C})[\alpha\times R(\theta^*(\overline{e_1})),\ldots,\alpha\times R(\theta^*(\overline{e_m}))] && \text{Définition de } R\\
&\leq \theta^*(\mathsf{C})[\alpha\times R(\alpha\times|[\![\overline{e_1}]\!]|),\ldots,\alpha\times R(\alpha\times|[\![\overline{e_m}]\!]|)] && \text{Lemme II.4}\\
&\leq Q(\alpha\times R(\alpha\times Q(\times \max_{i=1..n}(|u_i|)))) && \text{Définition de } Q\\
&\leq P'(\max_{i=1..n}(|u_i|))
\end{aligned}$$

et le nouveau max-polynôme $P'$ ainsi défini est donc une borne supérieure sur la taille des valeurs calculées. □

**Lemme II.48.** *Étant donné un programme arborescent et sympathique* $\boldsymbol{p}$ *et un arbre des appels de racine* $\langle \mathtt{f}, u_1,\cdots,u_n\rangle$ *correspondant à une exécution de* $\boldsymbol{p}$ *sur les entrées* $u_1,\cdots,u_n$*, alors la profondeur de l'arbre des appels est bornée par* $\gamma\times\log(\omega_{\mathtt{f}}(\theta^*(\overline{u})))$ *avec* $\gamma$ *constante.*

*Démonstration.* D'après le lemme II.45, on sait que la profondeur d'une branche d'états correspondant à des fonctions équivalentes est bornée par $\beta\times\log(\omega_{\mathtt{g}}(\theta^*(v_1),\ldots,\theta^*(v_k)))$, avec $\beta$ une constante, si $\langle \mathtt{g}, v_1,\cdots,v_k\rangle$ est le premier état de la branche. Les lemmes II.47 et II.4 impliquent que la profondeur d'une branche d'états correspondant à des fonctions équivalentes est bornée par $\beta\times\log(\omega_{\mathtt{g}}(\alpha\times P(|\overline{u}|)))$, si $\langle \mathtt{f}, u_1,\cdots,u_n\rangle$ est la racine de l'arbre des appels, $\alpha$ est la constante du lemme II.4 et $P$ est le polynôme du lemme II.47. Le nombre de classes d'équivalence pour la précédence $\approx_{Fct}$ étant borné par la taille du programme, on obtient le résultat désiré, pour une constante $\gamma$ choisie suffisamment grande mais indépendante des entrées. □

### II.4.3 Caractérisations de ALogTime, NC$^k$ et NC

Dans cette section, nous donnons une caractérisation des fonctions calculables dans $NC^k$ en utilisant les définitions de [Ruz81]. Il nous faut d'abord définir ce que nous entendons par fonction calculable dans $NC^k$ lorsque nous considérons des programmes

de notre langage. On code les éléments de $\mathcal{V}^*$ par une fonction *code* : $\mathcal{V}^* \rightarrow \{0,1\}^*$ calculable dans $NC^1$ et faisant correspondre à tout symbole de constructeur un circuit $U_{E^*}$-uniforme de taille polynomiale et de profondeur logarithmique. Étant donnés une fonction $\phi : \mathcal{V}^n \rightarrow \mathcal{V}$ calculée par un programme et un codage *code*, on leur associe une fonction $\tilde{\phi} : \{0,1\}^* \rightarrow \{0,1\}^*$ définie par $code(\phi(\overline{u})) = \tilde{\phi}(code(\overline{u}))$ pour toute valeur $u \in \mathcal{V}$. Une fonction $\phi$ de $\mathcal{V}$ est calculée dans $NC^k$ relativement à *code* si et seulement si $\tilde{\phi}$ est calculable par une famille $U_{E^*}$-uniforme de circuits dans $NC^k$.

À présent, il nous faut restreindre les opérateurs utilisés :

**Définition II.49.** *Un programme* $\mathbf{p}$ *a des* $k$*-opérateurs s'il existe un codage code tel que tout symbole d'opérateur de Op est calculé par une famille de circuits* $U_{E^*}$*-uniforme de taille polynomiale et de profondeur* $\log^k$ *relativement à code (C'est à dire* $NC^k$*).*

**Remarque 10.** *Les* 0*-opérateurs sont aussi appelés opérateurs plats (ou flat operator) dans [BMP06].*

**Exemple 34.** *L'opérateur* `if` *défini par* $[\![\texttt{if}]\!](u, u_0, u_1) = [\![u_i]\!]$ *suivant que* $[\![u]\!] = \mathbf{0}(\epsilon)$ *ou* $\mathbf{1}(\epsilon)$*, avec* $\epsilon$*,* $\mathbf{0}$ *et* $\mathbf{1}$ *des symboles de constructeur d'arités respectives* 0,1 *et* 1 *permettant de coder les mots binaires, est un* 0*-opérateur car le branchement peut être aisément codé par un circuit de taille polynomiale et de profondeur constante.*

Il nous faut maintenant définir un rang afin de distinguer les différents niveaux de la hiérarchie $NC^k$ :

**Définition II.50.** *Le rang d'un symbole* $b$*,* $\texttt{rk}(b)$*, et le rang d'un symbole* $b$ *relativement à une expression* $e$*,* $\nabla(b, e)$*, sont des fonctions à valeur dans* $\mathbb{N}$ *et définies par induction sur la précédence* $\geq_{Fct}$ *:*

- *Si* $b$ *est une variable ou un symbole de constructeur alors* $\texttt{rk}(b) = 0$.
- *Si* $b$ *est un* $k$*-opérateur alors* $\texttt{rk}(b) = k$.
- *Si* $b$ *est un symbole de fonction alors on définit son rang relativement à une expression par :*
  - *Si* $\forall \texttt{g} \in e$, $b >_{Fct} \texttt{g}$ *alors* $\nabla(b, e) = \max_{b' \in e}(\texttt{rk}(b'))$
  - *Sinon* $e = b'(e_1, \cdots, e_n)$ *et si :*
    - $b >_{Fct} b'$ *alors* $\nabla(b, e) = \max(\texttt{rk}(b') + 1, \nabla(b, e_1), \ldots, \nabla(b, e_n))$
    - $b \approx_{Fct} b'$ *alors* $\nabla(b, e) = \max(\nabla(b, e_1), \ldots, \nabla(b, e_n)) + 1$
- *Enfin, on définit le rang d'un symbole de fonction* $b$ *par :*
  - $\texttt{rk}(b) = \max(\nabla(b, e))$ *pour toute expression* $e$ *et pour tous motifs* $\overline{p}$ *tels que* $e$ *est une expression maximale activée par* $b(\overline{p})$

*Le rang d'un programme est défini comme étant le plus haut rang d'un symbole du programme. Le rang d'un état de l'arbre des appels est le rang du symbole de fonction correspondant à cet état. Le rang d'un arbre des appels est le plus haut rang d'un de ses états.*

**Exemple 35.** *Les opérateurs* **Left** *et* **Right** *sont des* 0*-opérateurs car calculables par un circuit de profondeur constante. Supposons que l'opérateur* $\odot$ *soit, lui aussi, un* 0*-opérateur, alors le programme de l'exemple 33 est de rang* 1*. En effet, le symbole de fonction* `sum` *est définie par une fraternité utilisant uniquement des symboles de rang* 0*.*

**Théorème II.51.** *Une fonction $\phi$ à valeur dans $\mathcal{V}^*$ est calculée par un programme sympathique et arborescent de rang $k \geq 1$ si et seulement si $\phi$ est calculée dans $NC^k$.*

*Démonstration.* Ce résultat est une conséquence directe des lemmes II.54 et II.60. □

Nous déduisons de ce résultat les caractérisations de *ALogTime* et de *NC* :

**Corollaire II.52.** *Une fonction $\phi$ à valeur dans $\mathcal{V}^*$ est calculée par un programme sympathique et arborescent de rang* 1 *si et seulement si $\phi$ est calculée dans ALogTime.*

**Corollaire II.53.** *Une fonction $\phi$ à valeur dans $\mathcal{V}^*$ est calculée par un programme sympathique et arborescent si et seulement si $\phi$ est calculée dans NC.*

**Remarque 11.** *La caractérisation de ALogTime a été introduite dans [BMP06]. La complétude y était démontrée de manière différente en utilisant une caractérisation de [LM00]. Ici, nous démontrons la complétude en utilisant la caractérisation de [Clo89].*

### Évaluation des programmes arborescents et sympathiques de rang k par des circuits

Nous présentons une implémentation des programmes par des familles uniformes de circuits. D'une part, nous avons établi une borne logarithmique sur la longueur de l'évaluation dans le lemme II.48 et nous avons démontré que la taille des valeurs calculées était bornée polynomialement en la taille des entrées II.47. D'autre part, nous allons construire le circuit $U_{E^*}$-uniforme de taille polynomiale et de profondeur logarithmique qui calcule un programme arborescent et sympathique de rang $k$.

**Lemme II.54.** *Étant donnée une fonction $\phi : \mathcal{V}^n \to \mathcal{V}$ définie par un programme arborescent et sympathique* $\mathbf{p}$ *de rang $k$, il existe une famille de circuits $U_{E^*}$-uniforme de taille polynomiale et de profondeur $\log^k$ qui calcule $\tilde{\phi}$.*

*Démonstration.* Étant donnée une borne supérieure $m$ sur la taille des entrées, nous allons construire le circuit $C_m$ correspondant à $m$ par induction sur le rang des symboles de fonction dans $\mathbf{p}$.

Soit $\mathtt{f}$ un symbole de fonction défini par un ensemble de $q$ définitions de la forme $\mathtt{f}(x_1, \cdots, x_n) = \textbf{Case } x_1, \cdots, x_n \textbf{ of } p_1, \cdots, p_n \to e_i$. Comme $\mathbf{p}$ est un programme orthogonal, les motifs contenus dans ces définitions ne se chevauchent pas. À partir des entrées, on construit un circuit discriminant qui va sélectionner la bonne définition de $\mathtt{f}$ à appliquer. Ce circuit est basé sur les différents codages des motifs et est défini par composition de sélecteurs (basés sur un circuit de type *if then else*) et de destructeurs calculés par des circuits de profondeur constante et de taille polynomiale.

Si $\mathtt{f}$ est de rang 0, alors les calculs de $\mathtt{f}$ n'impliquent que des symboles de constructeur et des 0-opérateurs, puisqu'aucune récurrence n'est autorisée. Par conséquent, le circuit correspondant est de profondeur constante et de taille polynomiale puisque le nombre d'opérateurs et de symboles de constructeur utilisé est borné par la taille du programme, donc par une constante.

À présent, supposons que $\mathtt{f}$ soit de rang $k$ et que tout symbole de fonction $\mathtt{g}$ de rang $j$ inférieur à $k$ tel que $\mathtt{f} >_{Fct} \mathtt{g}$ est calculé par une famille de circuits de profondeur $\log^j$ et de taille polynomiale. Il nous faut considérer deux cas de figure qui dépendent de la structure de $e_i$, lorsque $\mathtt{f}$ est défini par une définition de la forme $\mathtt{f}(x_1, \cdots, x_n) = \textbf{Case } x_1, \cdots, x_n \textbf{ of } p_1, \cdots, p_n \to e_i$.

Le premier cas à considérer est lorsque $e_i$ est construit à partir de symboles de fonction dont le rang est strictement inférieur au rang $k$ de $\mathtt{f}$. Par hypothèse d'induction, une famille de circuits de taille polynomiale et de profondeur au plus $\log^{k-1}$ calcule les symboles de fonctions dans $e_i$. On conclut en composant ces circuits avec le circuit discriminant. Le circuit obtenu est toujours de taille polynomiale et de profondeur bornée par $\log^k$.

Le second cas de figure advient lorsque $e_i = \mathsf{C}[\mathtt{g}_1(\overline{s_1}), \ldots, \mathtt{g}_r(\overline{s_r})]$ est une fraternité activée par $\mathtt{f}(p_1, \cdots, p_n)$.

Supposons que le contexte $\mathsf{C}[\diamond_1, \ldots, \diamond_r]$ soit construit à partir de symboles de fonction de rang strictement inférieur à $k$. Par hypothèse d'induction, on a donc un circuit de profondeur $\log^{k-1}$ et de taille polynomiale pour calculer $\mathsf{C}[\diamond_1, \ldots, \diamond_r]$. De même, on a un circuit de profondeur $\log^{k-1}$ et de taille polynomiale pour calculer les $(\overline{s_j})_{j=1,\ldots,r}$. Le circuit calculant la fraternité $e_i$ est construit en composant les entrées du circuit qui correspond à $\mathsf{C}[\diamond_1, \ldots, \diamond_r]$ avec les circuits qui correspondent aux $(\overline{s_j})_{j=1,\ldots,r}$. Le lemme II.48 nous garantit que la profondeur d'une telle composition est bornée par $d \times \log(m)$ avec $d$ une constante fixée et $m$ la taille de l'entrée. On obtient donc un arbre de sous-circuits de profondeur logarithmique. Le nombre de sous-circuits est donc polynomial. Pour s'en convaincre, il suffit de compter le nombre de noeuds contenu dans un arbre de profondeur logarithmique.

Supposons que le contexte comprenne des symboles de fonction de rang égal à $k$. Ceci signifie que le rang des symboles ne composant pas les $\mathtt{g}_i$ l'emporte strictement sur le rang des symboles composant les $\mathtt{g}_i$ dans l'appel récursif. Comme ces symboles ne sont pas composés, on les calcule en parallèle par un circuit de profondeur au plus $\log^k$ en appliquant l'hypothèse d'induction. Cette induction peut bien être appliquée puisque ces symboles sont strictement plus petits pour la précédence. Nous n'ajoutons ainsi aucune complexité supérieure à $\log^k$ dans la profondeur des circuits considérés. Pendant ce temps, on suit le même raisonnement que ci-dessus pour les autres symboles du circuits qui possède un rang strictement inférieur à $k$. Le circuit final a donc une profondeur $\log^k$ et une taille polynomiale.

Enfin, le lemme II.47 nous dit que la taille de chaque sous-circuit est bornée polynomialement en la taille des entrées. A fortiori, la nombre de portes d'entrée et de sortie de chaque sous-circuit est bornée polynomialement. Le nombre de sous-circuits étant polynomial, on obtient un circuit de taille polynomiale. Nous concluons en faisant remarquer que l'on compose au plus $d \times \log(m)$ sous-circuits qui, par hypothèse d'induction ont une profondeur $\log^{k-1}$, obtenant ainsi un circuit de profondeur au plus $O(\log^k)$. La condition d'$U_{E^*}$-uniformité est facile à vérifier puisque le langage étendu des connexions est basé sur le programme **p**, qui est fixé, ainsi que sur les bornes supérieures fournies par les lemmes II.48 et II.47.

□

### Simulation des fonctions calculables dans $\mathsf{NC}^k$

Il nous reste à démontrer qu'une fonction de $NC^k$ peut être calculée par un programme arborescent et sympathique de rang $k$. Dans cette optique, nous considérerons la caractérisation de [Clo89] plutôt que d'utiliser directement des Machines de Turing Alternantes (ATM). Nous procédons de cette manière pour deux raisons principales. La première est l'obtention d'une preuve simplifiée, car l'utilisation d'ATM nécessiterait un codage minutieux et très technique. La deuxième est encouragée par les liens ténus qui existent entre les schémas de récurrence utilisés dans l'étude de la complexité implicite des programmes et le langage de programmation fonctionnelle utilisé dans notre approche.

Dans [Clo89], la caractérisation des $NC^k$ est basée sur deux schémas de récurrence à la Cobham [Cob62] appelés « Concatenation Recursion on Notation »(CRN) et « Weak bounded Recursion on Notation »(WBRN) définis sur l'algèbre des fonctions des entiers naturels vers les entiers naturels en considérant les fonctions initiales suivantes $\mathtt{zero}(x) = 0$, $s_0(x) = 2\times x$, $s_1(x) = 2\times x+1$, $\pi^n_k(x_1, \cdots, x_n) = x_k$, $|x| = \lceil \log_2(x+1) \rceil$, $x\#y = 2^{|y|\times|x|}$ et $\mathtt{bit}(x, i) = \lfloor x/2^i \rfloor \bmod 2$.

Une fonction $\mathtt{f}$ est définie par CRN à partir des fonctions $\mathtt{g}$, $\mathtt{h}_0$ and $\mathtt{h}_1$ si $\forall n \in \mathbb{N},\ \forall x \in \mathbb{N},\ \mathtt{h}_i(n,x) \leq 1$ et on a :

$$\begin{aligned} \mathtt{f}(0,x) &= \mathtt{g}(x) \\ \mathtt{f}(s_0(n),x) &= s_{\mathtt{h}_0(n,x)}(\mathtt{f}(n,x)) && \text{if n} \neq 0 \\ \mathtt{f}(s_1(n),x) &= s_{\mathtt{h}_1(n,x)}(\mathtt{f}(n,x)) \end{aligned}$$

Une fonction $\mathtt{f}$ est définie par WBRN à partir des fonctions $\mathtt{g}$, $\mathtt{h}_0$, $\mathtt{h}_1$ et $\mathtt{r}$ si on a :

$$\begin{aligned} \mathtt{F}(0,x) &= \mathtt{g}(x) \\ \mathtt{F}(s_0(n),x) &= \mathtt{h}_0(n,x,\mathtt{F}(n,x)) && \text{if n} \neq 0 \\ \mathtt{F}(s_1(n),x) &= \mathtt{h}_1(n,x,\mathtt{F}(n,x)) \\ \mathtt{f}(n,x) &= \mathtt{F}(|n|,x) \\ \forall n \in \mathbb{N},\ \forall x \in \mathbb{N},\ \mathtt{f}(n,x) &\leq \mathtt{r}(n,x) \end{aligned}$$

**Définition II.55.** *La fonction* $\mathtt{tree}$ *est définie par le schéma suivant :*

$$\begin{aligned} &\mathtt{and}(0) = 0 && \mathtt{or}(0) = 0 \\ &\mathtt{and}(1) = 0 && \mathtt{or}(1) = 1 \\ &\mathtt{and}(s_0(s_0(x))) = s_0(\mathtt{and}(x)) && \mathtt{or}(s_0(s_0(x))) = s_0(\mathtt{or}(x)) \\ &\mathtt{and}(s_0(s_1(x))) = s_0(\mathtt{and}(x)) && \mathtt{or}(s_0(s_1(x))) = s_1(\mathtt{or}(x)) \\ &\mathtt{and}(s_1(s_0(x))) = s_0(\mathtt{and}(x)) && \mathtt{or}(s_1(s_0(x))) = s_1(\mathtt{or}(x)) \\ &\mathtt{and}(s_1(s_1(x))) = s_1(\mathtt{and}(x)) && \mathtt{or}(s_1(s_1(x))) = s_1(\mathtt{or}(x)) \\ &\mathtt{contract}(x) = \mathtt{or}(\mathtt{and}(x)) \end{aligned}$$

$$\mathtt{tree}(x) = \ \textit{si } |x| < 4 \textit{ alors } x \mathtt{\ mod\ } 2 \textit{ sinon } \mathtt{tree}(\mathtt{contract}(x))$$

**Définition II.56.** *Le Rang de Clote* Crk *d'une fonction est défini de la manière suivante :*

- *Le rang* Crk *de la fonction* $\mathtt{tree}$ *et des fonctions initiales est* 0.
- *Si* $\mathtt{f}$ *est définie par composition de* $\mathtt{f}_1, \cdots, \mathtt{f}_n$ *alors :*
$$\text{Crk}(\mathtt{f}) = \max(\text{Crk}(\mathtt{f}_1), \ldots, \text{Crk}(\mathtt{f}_n))$$
- *Si* $\mathtt{f}$ *est définie par CRN à partir de* $\mathtt{h}_0,\ \mathtt{h}_1$ *et* $\mathtt{g}$ *alors :*
$$\text{Crk}(\mathtt{f}) = \max(\text{Crk}(\mathtt{h}_0), \text{Crk}(\mathtt{h}_1), \text{Crk}(\mathtt{g}))$$
- *Enfin, si* $\mathtt{f}$ *est définie par WBRN à partir de* $\mathtt{h}_0,\ \mathtt{h}_1,\ \mathtt{g}$ *et* $\mathtt{r}$ *alors :*
$$\text{Crk}(\mathtt{f}) = \max(1 + \text{Crk}(\mathtt{h}_0), 1 + \text{Crk}(\mathtt{h}_1), \text{Crk}(\mathtt{g}), \text{Crk}(\mathtt{r}))$$

**Définition II.57.** *L'algèbre $N$ est définie comme la classe des fonctions contenant les fonctions initiales, la fonction* `tree` *et close par $CRN$, par $WBRN$ et par composition. $N_k$ est la sous-classe de $N$ des fonctions* `f` *de rang* $\text{Crk}(\mathtt{f})$ *inférieur ou égal à $k$.*

**Théorème II.58** (Clote [Clo89])**.**

$$\forall k \geq 0,\ N_k = NC^{k+1}$$

Nous allons utiliser ce résultat afin de démontrer la complétude de notre caractérisation. En premier lieu, nous simulons les fonctions initiales par des 1-opérateurs. Cette simulation peut être effectuée sans problème puisque les fonctions initiales sont toutes dans la classe $AC^0$ qui est incluse dans $NC^1$. Il reste à vérifier que l'on peut simuler la fonction `tree` par un symbole de fonction correspondant à un programme arborescent et sympathique de rang 1 à valeur sur les mots binaires $\mathtt{W} = \{0,1\}^*$, où les bits 0 et 1 sont représentés à l'aide d'un codage *code* par les symboles de constructeur **0** et **1** et où le mot vide est codé par le symbole d'arité nulle $\epsilon$. Dans la suite de cette section, étant donnée une fonction `f` définie dans l'algèbre de Clote, $\hat{\mathtt{f}}$ représente le symbole de fonction de notre langage simulant `f`, c'est-à-dire $\forall \overline{n} \in \overline{\mathtt{W}},\ code(\mathtt{f}(\overline{n})) = \hat{\mathtt{f}}(code(\overline{n}))$.

**Lemme II.59.** *La fonction* `tree` *peut être simulée par un programme arborescent et sympathique* $\widehat{\mathtt{tree}}$ *de rang* 1 *sur les mots binaires* `W`*.*

*Démonstration.* Supposons que la fonction log soit simulée par le 1-opérateur $\widehat{\log}$ et que les fonctions `and` et `or` soient simulées par des 0-opérateurs $\widehat{\mathtt{and}}$ et $\widehat{\mathtt{or}}$. Remarquons qu'une telle simulation est facile puisque ces fonctions calculent la conjonction ainsi que la disjonction des bits deux à deux tandis que la fonction log peut être calculée par un programme arborescent et sympathique de rang 1. Nous omettons ce programme avec comme objectif de simplifier la preuve.

On définit la fonction `g` par $\mathtt{g}(y) = \textbf{Case } y \textbf{ of } \mathbf{i}(\mathbf{j}(x)) \rightarrow \widehat{\mathtt{or}}(\widehat{\mathtt{and}}(\mathbf{i}(\mathbf{j}(x))))$ en supposant que les sup-interprétations des symboles de constructeur sont additives et égales à $\theta(\mathbf{i})(X) = X+1, \forall \mathbf{i}, \mathbf{j} \in \{\mathbf{0}, \mathbf{1}\}$ et $\theta(\epsilon) = 0$. Une telle sup-interprétation a pour propriété que la sup-interprétation d'un mot est égale à sa taille, puisque tout symbole d'arité strictement positive ajoute 1 à la valeur de la sup-interprétation. Les symboles d'opérateur $\widehat{\mathtt{or}}$ et $\widehat{\mathtt{and}}$ retournent $\lceil n/2 \rceil$ bits de sortie si $n$ bits sont donnés en entrée. Par conséquent, la composition $\widehat{\mathtt{or}}(\widehat{\mathtt{and}})$ de ces deux fonctions retourne $\lceil \lceil n/2 \rceil /2 \rceil$ bits en sortie si $n$ bits lui sont fournis en entrée. Remarquons que le symbole de fonction `g` peut seulement être appliqué pour $n \geq 2$ puisque le motif définissant `g` est de taille supérieure ou égale à deux. Puisque $\forall n \geq 2, \lceil \lceil n/2 \rceil /2 \rceil \leq n/2$, on peut choisir la sup-interprétation

de $\mathtt{g}$ comme étant égale à $\theta(\mathtt{g})(X) = X/2$. Il nous reste à définir le symbole de fonction $\hat{\mathtt{tree}}$ simulant la fonction $\mathtt{tree}$ :

$$\hat{\mathtt{tree}}(y) = \textbf{Case } y \textbf{ of } \mathbf{i}(x) \to \mathbf{if}(\mathtt{test}(\hat{\log}(\mathbf{i}(x))), \mathtt{mod}_2(\mathbf{i}(x)), \hat{\mathtt{tree}}(\mathtt{g}(\mathbf{i}(x)))) \quad i \in \{\mathbf{0}, \mathbf{1}\}$$

où $\mathtt{test}$ est un 0-opérateur qui, sur une entrée $y$, teste si la taille de $y$ est strictement inférieure à 4 et où $\mathtt{mod}_2$ est un 0-opérateur qui rend le dernier bit d'un mot donné en entrée.

En choisissant $\omega_{\hat{\mathtt{tree}}}(X) = X$, on peut d'abord montrer que le programme est arborescent :

$$\begin{aligned}
\omega_{\hat{\mathtt{tree}}}(\theta^*(\mathbf{i}(x))) &= \omega_{\hat{\mathtt{tree}}}(X+1) \\
&= X+1 \\
&= K \times \omega_{\hat{\mathtt{tree}}}(\theta^*(\mathtt{g}(\mathbf{i}(x)))) \qquad \text{avec K=2}
\end{aligned}$$

De plus, en prenant $\theta(\mathbf{if})(X,Y,Z) = \max(Y,Z)$, $\theta(\mathtt{test})(X) = 1$ et $\theta(\mathtt{mod}_2)(X) = 1$, on vérifie aisément que le programme est sympathique. $\hat{\mathtt{tree}}$ est défini par la fraternité $e = \mathsf{C}[\mathtt{g}(\mathbf{i}(x))]$ où le contexte $\mathsf{C}[\diamond]$ est égal à $\mathbf{if}(\mathtt{test}(\hat{\log}(\mathbf{i}(x))), \mathtt{mod}_2(\mathbf{i}(x)), \hat{\mathtt{tree}}(\diamond))$. Il en résulte, par définition du rang, que :

$$\begin{aligned}
\mathtt{rk}(\hat{\mathtt{tree}}) &= \nabla(\hat{\mathtt{tree}}, \mathbf{if}(\mathtt{test}(\hat{\log}(\mathbf{i}(x))), \mathtt{mod}_2(\mathbf{i}(x)), \hat{\mathtt{tree}}(\mathtt{g}(\mathbf{i}(x))))) \\
&= \max(\mathtt{rk}(\mathbf{if}) + 1, \nabla(\hat{\mathtt{tree}}, \mathtt{test}(\hat{\log}(\mathbf{i}(x)))), \nabla(\hat{\mathtt{tree}}, \mathtt{mod}_2(\mathbf{i}(x))), \nabla(\hat{\mathtt{tree}}, \\
&\hat{\mathtt{tree}}(\mathtt{g}(\mathbf{i}(x))))) \\
&= \max(1, \mathtt{rk}(\mathtt{test}), \mathtt{rk}(\hat{\log}), \mathtt{rk}(\mathbf{i}), \mathtt{rk}(\mathtt{mod}_2), \nabla(\hat{\mathtt{tree}}, \mathtt{g}(\mathbf{i}(x))) + 1) \\
&= \max(1, \mathtt{rk}(\mathtt{g}) + 1) \\
&= \max(1, 0+1) = 1
\end{aligned}$$

et nous obtenons le résultat énoncé. □

**Lemme II.60.** *Une fonction $\phi$ de $N_k$ est calculable par un programme arborescent et sympathique $\boldsymbol{p}$ de rang au plus $k+1$.*

*Démonstration.* Nous essayons de simuler toutes les fonctions de $N_k$ par des programmes arborescents et sympathiques de rang $k+1$ à valeur sur les mots binaires $\mathtt{W} = \{0,1\}^*$.

Les fonctions initiales ainsi que la fonction $\mathtt{tree}$ peuvent être simulées par des programmes arborescents et sympathique de rang au plus 1 comme nous l'avons démontré dans le lemme II.59.

Étant donnée une fonction $\mathtt{f}$ définie par CRN à partir des fonctions $\mathtt{g}$, $\mathtt{h_0}$ et $\mathtt{h_1}$. Par hypothèse d'induction, supposons que nous ayons déjà défini les programmes $\hat{\mathtt{g}}$, $\hat{\mathtt{h_0}}$ et $\hat{\mathtt{h_1}}$

qui calculent ces fonctions. On simule le schéma CRN par le programme suivant pour $\mathbf{i},\mathbf{j},\mathbf{i}' \in \{\mathbf{0},\mathbf{1}\}$ :

$$\hat{\mathtt{f}}(n,x) = \mathtt{rev}(\mathbf{conc}(\mathtt{h}(n,x,n)),\mathtt{rev}(\hat{\mathtt{g}}(x)))$$

$$\mathtt{h}(\overline{u}) = \mathbf{Case}\ \overline{u}\ \mathbf{of}\ \mathbf{i}(\mathbf{j}((n))),x,\mathbf{i}'(n') \to \mathsf{C}[(\mathtt{h}(\mathbf{Fh}(\mathbf{j}(n)),x,n'),\mathtt{h}(\mathbf{Sh}(\mathbf{j}(n)),x,\mathbf{Sh}(\mathbf{j}(n))))]$$

$$\mathbf{i}(\epsilon),x,\mathbf{j}(n) \to \mathtt{if}(\hat{\mathtt{h}_\mathbf{j}}(n,x),\mathbf{0}(\epsilon),\mathbf{1}(\epsilon))$$

avec $\mathsf{C}[\diamond_1,\diamond_2] = \mathbf{if}(\hat{\mathtt{h}_{\mathbf{i}'}}(n',x),\mathbf{0}(\mathbf{conc}(\diamond_1,\diamond_2)),\mathbf{1}(\mathbf{conc}(\diamond_1,\diamond_2)))$ et où $\mathbf{Fh}(x)$ et $\mathbf{Sh}(x)$ sont des 0-opérateurs calculant les $\lfloor x/2 \rfloor$ premiers bits et, respectivement, les $\lceil x/2 \rceil$ derniers bits de $x$. Tous ces opérateurs sont facilement calculables : Ce sont bien des 0-opérateurs car il a été démontré dans [Blo94] qu'ils sont calculables dans $NC^0$. **conc** représente un 0-opérateur qui calcule la concaténation de deux mots tandis que `rev` est un 0-opérateur qui renverse un mot.

Nous allons expliquer brièvement le calcul effectué par ce programme. Le symbole de fonction $\hat{\mathtt{f}}$ sert juste à initialiser le calcul en lançant la récurrence à l'aide de l'appel du symbole de fonction `h`, puis, concatène le résultat du calcul de la récurrence avec le résultat retourné par la fonction de base $\hat{\mathtt{g}}$. L'opérateur `rev` est utilisé car les calculs sont toujours codés modulo le sens de lecture des mots. Le symbole de fonction `h` simule le schéma CRN. Cependant, au lieu d'être appelé linéairement sur l'entrée, il s'exécute de manière dichotomique en divisant les données en deux et effectue, à chaque pas, une concaténation des résultats retournés. Tandis que le premier argument de `h` sert à effectuer la division des données en deux, le troisième argument sert à stocker l'entrée sur laquelle le programme travaille.

En prenant $\theta(\mathbf{Fh})(X) = \lfloor X/2 \rfloor$ , $\theta(\mathbf{Sh})(X) = \lceil X/2 \rceil$, $\theta(\mathbf{i})(X) = \theta(\mathbf{j})(X) = X+1$, pour $\mathbf{i},\mathbf{j} = \mathbf{0},\mathbf{1}$ et $\omega_{\mathtt{h}}(N,X,N') = N$, on peut montrer que le programme est arborescent :

$$\begin{aligned}
\omega_{\mathtt{h}}(\theta^*(\mathbf{i}(\mathbf{j}(n))),\theta(x),\theta^*(\mathbf{i}'(n'))) &= N+2 \\
&\geq K \times \lceil (N+1)/2 \rceil \qquad \text{avec K=2} \\
&= K \times \omega_{\mathtt{h}}(\theta^*(\mathbf{Sh}(\mathbf{j}(n))),\theta(x),\theta^*(\mathbf{Sh}(\mathbf{j}(n)))) \\
&\geq K \times \omega_{\mathtt{h}}(\theta^*(\mathbf{Fh}(\mathbf{j}(n))),\theta(x),\theta^*(n'))
\end{aligned}$$

En prenant $\theta(\mathbf{if})(X,Y,Z) = \max(Y,Z)$, $\theta(\hat{\mathtt{h}_\mathbf{i}})(N,X) = 1$, puisque $\mathtt{h}_\mathbf{i}$ ne renvoie qu'un seul bit en sortie, et $\theta(\mathbf{conc})(X,Y) = X+Y$, on démontre que le programme est sympathique :

$$\theta^*(\mathsf{C}[\diamond_1,\diamond_2]) = \theta(\diamond_1) + \theta(\diamond_2) + 1$$

$$\theta(\mathbf{Fh})(N) \leq \theta(\mathbf{Sh})(N) \leq N$$

Par définition du rang de Clote, on sait que :

$$\mathrm{Crk}(\mathtt{f}) = \max(\mathrm{Crk}(\mathtt{h_0}), \mathrm{Crk}(\mathtt{h_1}), \mathrm{Crk}(\mathtt{g})) = k$$

Par hypothèse d'induction, $\mathtt{h_0}, \mathtt{h_1}$ et $\mathtt{g}$ peuvent être calculées par des programmes arborescents et sympathiques $\hat{\mathtt{g}}$, $\hat{\mathtt{h_0}}$ et $\hat{\mathtt{h_1}}$ de rangs bornés respectivement par $k+1$. Nous calculons le rang du programme $\hat{\mathtt{f}}$ :

$$\begin{aligned}
\mathtt{rk}(\hat{\mathtt{f}}) &= \nabla(\hat{\mathtt{f}}, \mathtt{rev}(\mathbf{conc}(\mathtt{h}(n,x,n)), \mathtt{rev}(\hat{\mathtt{g}}(x)))) \\
&= \max(\mathtt{rk}(\mathtt{h}), \mathtt{rk}(\hat{\mathtt{g}}), \mathtt{rk}(\mathtt{rev}), \mathtt{rk}(\mathbf{conc})) \\
&\leq \max(\mathtt{rk}(\hat{\mathtt{g}}), \nabla(\mathtt{h}, \mathsf{C}[(\mathtt{h}(\mathbf{Fh}(\mathbf{j}(n)), x, n'), \mathtt{h}(\mathbf{Sh}(\mathbf{j}(n)), x, \mathbf{Sh}(\mathbf{j}(n))))])) \\
&\leq \max(\mathtt{rk}(\hat{\mathtt{g}}), \mathtt{rk}(\hat{\mathtt{h}_i}), \max(\mathtt{rk}(\mathbf{Fh}), \mathtt{rk}(\mathbf{Sh}), \mathtt{rk}(\mathbf{0}), \mathtt{rk}(\mathbf{1}), \mathtt{rk}(\mathbf{if}), \mathtt{rk}(\mathbf{conc})) + 1) \\
&= \max(k+1, 0+1) \\
&= k+1 = \mathrm{Crk}(\mathtt{f}) + 1
\end{aligned}$$

et obtenons le résultat requis.

Maintenant, supposons que la fonction $\mathtt{f}$ soit définie par WBRN à partir des fonctions $\mathtt{g}$, $\mathtt{h_0}$, $\mathtt{h_1}$ et $\mathtt{r}$. Par hypothèse d'induction, supposons que nous ayons déjà défini les programmes correspondants dont les symboles de fonctions sont respectivement $\hat{\mathtt{g}}$, $\hat{\mathtt{h_0}}$, $\hat{\mathtt{h_1}}$ et $\hat{\mathtt{r}}$. En outre, nous supposerons que les $\hat{\mathtt{h}_i}$ vérifient pour touts mots binaires $r, u, x, t$ $[\![\hat{\mathtt{h}_i}]\!](r,u,x,t) \leq r$. On va simuler le schéma de récurrence WBRN par le programme suivant, pour $\mathbf{i}, \mathbf{j} \in \{\mathbf{0}, \mathbf{1}\}$ :

$$\begin{aligned}
\hat{\mathtt{f}}(n,x) &= \mathtt{F}(n, x, \hat{\mathtt{r}}(n,x), \log(n)) \\
\mathtt{F}(\overline{u}) = \textbf{Case } \overline{u} \textbf{ of } \mathbf{i}(n), x, r, \mathbf{j}(u) &\rightarrow \mathtt{if}(\mathtt{test}(u), \\
&\qquad \hat{\mathtt{h}_1}(r, u, x, \mathtt{F}(\mathbf{Fh}(n), x, r, u)), \\
&\qquad \hat{\mathtt{h}_0}(r, u, x, \mathtt{F}(\mathbf{Fh}(n), x, r, u))) \\
\epsilon, x, r, u &\rightarrow \hat{\mathtt{g}}(x)
\end{aligned}$$

où $\mathtt{test}(u)$ est un 0-opérateur qui retourne $\mathbf{0}(\epsilon)$ si le dernier bit de $u$ est 1 et $\mathbf{1}(\epsilon)$ autrement. De plus remarquons que tous les opérateurs utilisés ci-dessus sont de rang inférieur ou égal à 1.

On constate que la taille de la valeur calculée par les fonctions $\hat{\mathtt{h}_i}$ est toujours inférieure à la taille de l'argument d'entrée $r$, par définition du schéma WBRN, et on peut donc choisir la sup-interprétation suivante $\theta(\hat{\mathtt{h}_i})(R,N,X,T) = R$ et $\theta(\mathbf{0})(X) = \theta(\mathbf{1})(X) = X+1$, puisque la sup-interprétation d'un mot est égale à sa taille.

Afin de donner une intuition au lecteur, nous expliquons ici le rôle des différents arguments $n, x, r, u$ du symbole de fonction $\texttt{F}$ :

- $n$ est l'argument sur lequel a lieu la récurrence. Sa taille est divisée par deux à chaque appel récursif et permet ainsi de borner le nombre d'appels récursifs successifs par un logarithme.
- $x$ représente les autres arguments utilisés dans le schéma de récurrence WBRN.
- $r$ est utilisé comme un emplacement mémoire qui stocke la borne supérieure fournie par la fonction $\hat{\texttt{r}}$ lorsque le programme est exécuté pour la première fois.
- $u$ est utilisé comme un registre stockant la valeur du logarithme de $n$.

En prenant $\omega_{\texttt{F}}(N, X, R, U) = N$ et $\theta(\mathbf{Fh})(N) = \lfloor N/2 \rfloor$, on démontre que le programme est arborescent :

$$\begin{aligned}\omega_{\texttt{F}}(\theta^*(\mathbf{i}(n)), \theta(x), \theta(r), \theta^*(\mathbf{j}(u))) &= N + 1 \\ &\geq K \times \lfloor N/2 \rfloor \qquad \text{avec K=2} \\ &= K \times \omega_{\texttt{F}}(\theta^*(\mathbf{Fh}(n)), \theta(x), \theta(r), \theta(u))\end{aligned}$$

Puis, on montre qu'il est sympathique :

$$\begin{aligned}\theta^*(\hat{\texttt{h}_i})(R, N, X, T) &= R \\ \theta(\mathbf{Fh})(N) &\leq N\end{aligned}$$

Il nous reste à vérifier qu'une fonction $\texttt{f}$ définie dans l'algèbre de Clote et vérifiant $\text{Crk}(\texttt{f}) = k$ peut être simulée par un programme arborescent et sympathique $\mathbf{p}$ de rang $\texttt{rk}(\mathbf{p}) \leq k + 1$. Supposons que $\texttt{f}$ est défini par WBRN à partir des fonctions $\texttt{h}_\texttt{0}, \texttt{h}_\texttt{1}, \texttt{g}$ et $\texttt{r}$ de rangs de Clote respectifs Crk $k_{\texttt{h}_\texttt{0}}, k_{\texttt{h}_\texttt{1}}, k_{\texttt{g}}$ et $k_{\texttt{r}}$. Par définition de Crk, on a $\text{Crk}(\texttt{f}) = \max(k_{\texttt{r}}, k_{\texttt{g}}, \max(k_{\texttt{h}_\texttt{1}}, k_{\texttt{h}_\texttt{0}}) + 1)$. Par hypothèse d'induction, ces fonctions peuvent être simulées par des programmes arborescents et sympathiques $\hat{\texttt{h}_\texttt{0}}, \hat{\texttt{h}_\texttt{1}}, \hat{\texttt{g}}$ et $\hat{\texttt{r}}$ de rangs bornés respectivement par $k_{\texttt{h}_\texttt{0}} + 1, k_{\texttt{h}_\texttt{1}} + 1, k_{\texttt{g}} + 1$ et $k_{\texttt{r}} + 1$. En partant du programme décrit ci-dessus, on dérive :

$$\begin{aligned}\texttt{rk}(\hat{\texttt{f}}) &\leq \max(\texttt{rk}(\texttt{r}), \texttt{rk}(\texttt{F})) \\ &= \max(k_{\texttt{r}} + 1, \max(\texttt{rk}(\texttt{g}), \nabla(\texttt{F}, \texttt{test}(u)), \max(\texttt{rk}(\hat{\texttt{h}_\texttt{0}}), \texttt{rk}(\hat{\texttt{h}_\texttt{1}})) + 1)) \\ &\leq \max(k_{\texttt{r}} + 1, k_{\texttt{g}} + 1, 1, k_{\texttt{h}_\texttt{0}} + 2, k_{\texttt{h}_\texttt{1}} + 2) \\ &= \text{Crk}(\texttt{f}) + 1\end{aligned}$$

et nous obtenons le résultat requis. □

## II.5 Synthèse de sup-interprétations

### II.5.1 Synthèse avec propriété sous-terme

Le critère quasi-amical défini en section II.2.1 permet d'obtenir des heuristiques afin de synthétiser des sup-interprétations. En effet, si l'on possède une sup-interprétation des symboles de constructeur et que l'on vérifie le critère pour les plus petits symboles de fonction pour la précédence $\geq_{Fct}$, alors on arrive à synthétiser un poids pour chacun de ces symboles de fonction. Le poids est une sup-interprétation du symbole de fonction modulo quelques compositions de polynômes. En itérant le procédé sur la précèdence, on arrive ainsi à synthétiser des sup-interprétations pour tout symbole de fonction. Malheureusement, le poids se doit de posséder la propriété sous-terme et l'on en revient ainsi à synthétiser des quasi-interprétations afin de trouver une sup-interprétation. Il nous faut donc trouver des heuristiques plus performantes permettant de trouver des fonctions sans la propriétés sous-terme.

### II.5.2 Synthèse sans propriété sous-terme

Dans cette section, nous allons réutiliser la notion de paire de dépendance et la notion d'assignation polynomiale afin de dériver un critère développé dans [MP07b] et permettant la synthèse de sup-interprétations. Une assignation vérifiant ce critère sera appelée interprétation d'espace polynomial et nous fournira une sup-interprétation, ne satisfaisant pas obligatoirement la propriété sous-terme. Nous éliminons la propriété sous-terme en la remplaçant par un quasi-ordre $\geq_\beta$ sur les polynômes inspiré par [Luc05]. En utilisant à nouveau les résultats sur la synthèse de quasi-interprétation obtenus dans la section I.3, on obtient que ce critère est décidable et nous obtenons, par conséquent, des heuristiques afin de trouver une sup-interprétation pour un programme donné.

**Définition II.61** ([Luc05])**.** *Étant donnée une constante $\beta > 0$, on définit la relation d'ordre stricte $\geq_\beta$ à valeur dans $\mathbb{R}$ par :*

- *$x >_\beta y$ si et seulement si $x \geq \beta + y$*

**Définition II.62** (Interprétation d'espace polynomial)**.** *Un programme **p** admet une interprétation d'espace polynomial s'il existe une assignation additive, monotone et polynomiale $(\!|-|\!)$ qui vérifie :*

- *Pour toute expression maximale e activée par $\mathtt{f}(p_1, \cdots, p_n)$, $(\!|\mathtt{f}(p_1, \cdots, p_n)|\!)^* \geq (\!|e|\!)^*$.*
- *Pour toute paire de dépendance $\langle s, t\rangle$, $(\!|s|\!)^* \geq (\!|t|\!)^*$*
- *Pour tout cycle dans le graphe des paires de dépendance, il existe une paire de dépendance $\langle s, t\rangle$ impliquée dans le cycle et telle que $(\!|s|\!)^* >_\beta (\!|t|\!)^*$*

**Exemple 36.** *Le programme de l'exemple 30 admet l'interprétation d'espace polynomial suivante :*

$$(\!|\mathbf{0}|\!) = 0 \qquad (\!|\mathbf{S}|\!)(X) = X+1$$
$$(\!|\mathtt{half}|\!)(X) = X/2 \qquad (\!|\mathtt{log}|\!)(X) = X$$

*En effet, pour chaque règle, on vérifie que :*

$$\begin{aligned}
(\!|\mathtt{half}(\mathbf{0})|\!)^* &= 0 \geq 0 = (\!|\mathbf{0}|\!)^* \\
(\!|\mathtt{half}(\mathbf{S}(\mathbf{0}))|\!)^* &= 1/2 \geq 0 = (\!|\mathbf{0}|\!)^* \\
(\!|\mathtt{half}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= (Y+2)/2 \geq Y/2+1 = (\!|\mathbf{S}(\mathtt{half}(y))|\!)^* \\
(\!|\mathtt{log}(\mathbf{0})|\!)^* &= 0 \geq 0 = (\!|\mathbf{0}|\!)^* \\
(\!|\mathtt{log}(\mathbf{S}(\mathbf{0}))|\!)^* &= 1 \geq 0 = (\!|\mathbf{0}|\!)^* \\
(\!|\mathtt{log}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= Y+2 \geq 2+Y/2 = (\!|\mathbf{S}(\mathtt{log}(\mathtt{half}(\mathbf{S}(\mathbf{S}(y)))))|\!)^*
\end{aligned}$$

*De plus, en choisissant $\beta = 1$, pour toute paire de dépendance, on a :*

$$\begin{aligned}
(\!|\mathtt{half}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= Y/2+1 >_\beta Y/2 = (\!|\mathtt{half}(y)|\!)^* \\
(\!|\mathtt{log}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= Y+2 >_\beta 1+Y/2 = (\!|\mathtt{log}(\mathtt{half}(\mathbf{S}(\mathbf{S}(y))))|\!)^* \\
(\!|\mathtt{log}(\mathbf{S}(\mathbf{S}(y)))|\!)^* &= Y+2 \geq Y/2 = (\!|\mathtt{half}(y)|\!)^*
\end{aligned}$$

*Remarquons que les inégalités correspondant aux cycles dans le graphe des paires de dépendance sont strictes.*

Tout d'abord, constatons qu'un programme dont le graphe des paires de dépendance admet une interprétation d'espace polynomial termine :

**Théorème II.63.** *Supposons que* $\boldsymbol{p}$ *soit un programme dont le graphe des paires de dépendance admet une interprétation d'espace polynomial* $(\!|-|\!)$*, alors le programme* $\boldsymbol{p}$ *termine.*

*Démonstration.* Ce résultat est une application directe des paires de dépendance en définissant le quasi-ordre $s \geq_{q.o.} t$ sur les termes comme étant égal à $(\!|s|\!)^* \geq (\!|t|\!)^*$. Par définition d'une interprétation d'espace polynomial, le critère de terminaison des paires de dépendance introduit en section II.3.2 est vérifié. Il reste à voir que le quasi-ordre considéré est monotone, bien-fondé et clos par substitution. Il est bien-fondé puisque l'inégalité stricte $>_\beta$ garantit une décroissance d'au moins une constante $\beta > 0$ fixée à l'avance à chaque application d'un appel récursif. Il est clos par substitution et monotone par définition des assignations. □

On montre que la taille des valeurs calculées reste bornée polynomialement :

**Proposition II.64.** *Étant donné un programme* **p** *qui admet une interprétation d'espace polynomial* $(\!|-|\!)$*, alors pour tout expression* $t$ *et pour toute substitution* $\sigma$ *telles que* $t\sigma \downarrow u$*, on a :*

$$(\!|t\sigma|\!)^* \geq (\!|u|\!)^*$$

*Démonstration.* Ce résultat est une conséquence directe de la proposition I.19. □

À présent, nous démontrons qu'une interprétation d'espace polynomial fournit une sup-interprétation polynomiale.

**Théorème II.65.** *Toute interprétation d'espace polynomial est une sup-interprétation.*

*Démonstration.* D'après la proposition I.19, pour toute interprétation d'espace polynomial, on a $(\!|\mathtt{f}(v_1, \cdots, v_n)|\!)^* \geq (\!|[\![\mathtt{f}(v_1, \cdots, v_n)]\!]|\!)^*$. Puisque l'interprétation d'espace polynomial est une assignation additive, le lemme II.4 implique que pour toute valeur $v$, $(\!|v|\!)^* \geq |v|$. Finalement, il nous reste à vérifier que toute interprétation d'espace polynomial est constituée d'assignations monotones. □

Une conséquence directe de ce théorème est l'obtention d'une procédure afin de calculer des sup-interprétations polynomiales non sous-terme. En effet, si l'on arrive à trouver une interprétation d'espace polynomial alors on obtient une sup-interprétation. De plus, cette procédure est décidable, pour peu que la constante $\beta$ soit fixée et que le degré des polynômes soient bornés, puisqu'elle ne consiste qu'en un système d'inégalités à valeur sur les réels similaires à celle de la section I.3.

## II.6 Conclusion

La sup-interprétation constitue une évolution importante dans l'étude des interprétations sémantiques des programmes puisqu'elle permet d'étudier la complexité d'un grand nombre d'algorithmes. En particulier, la sup-interprétation est un outil plus puissant que la quasi-interprétation puisqu'il analyse un nombre strictement plus grand de programmes. De plus, elle permet aussi, comme nous l'avons vu, de caractériser les classes de complexité polynomiales avec une plus grande complétude intensionnelle. La sup-interprétation s'avère être un outil plus fin qui permet de caractériser des petites classes de complexité parallèles. Enfin, nous avons montré que l'on pouvait synthétiser des sup-interprétations en utilisant les quasi-interprétations ou des interprétation d'espace polynomial.

Les sup-interprétations offrent un avantage supplémentaire en terme de flexibilité par rapport au langage étudié. En effet, elles permettent de considérer des opérateurs ; c'est-

à-dire des opérations primitives. Cette particularité nous encourage à adapter l'outil à des langages de programmation orientés objet.

# III Extension des sup-interprétations à un langage orienté objet

Quand il y a plus de vingt ans,
nous jouions avec fièvre au premier ping-pong électronique,
pensions-nous une seconde que nous serions ridicules
aux yeux de la nouvelle génération,
pitoyables pour la génération suivante et misérables pour celle d'après ?
Mais la fierté d'avoir été les premiers Homo Electronicus
ne nous sera jamais enlevée.
Pionniers nous fûmes, et pionniers nous resterons.
Nous avons su où se dessinait l'Avenir.
Une humanité où tous les hommes regardent enfin
dans la même direction,
celle de leur écran d'ordinateur.

---

*(Lewis Trondheim, Ordinateur mon ami)*

Comme nous l'avons vu dans les deux précédents chapitres, une sup-interprétation est un outil introduit afin d'étudier la complexité implicite des langages de programmation fonctionnels au premier ordre. Une sup-interprétation fournit une borne supérieure sur la taille des calculs effectués par un symbole de fonction. En outre, elle permet de considérer un langage fonctionnel incluant des opérateurs. Les opérateurs sont des opérations primitives du langage dont la description est fournie par le créateur du langage. Étant donnés un opérateur et sa sup-interprétation, cette dernière garantit que le calcul sera effectué par l'opérateur en utilisant une quantité de ressources fixée à l'avance. Si l'obtention de tels certificats est une propriété indécidable en général, elle permet néanmoins de considérer des opérations dont le calcul n'est pas déterminé par le programmeur en personne mais plutôt par les spécifications du langage. Cette flexibilité supplémentaire nous suggère fortement d'adapter un tel outil à des langages de programmation de plus

haut niveau comme des langages impératifs ou orientés objet qui sont construits à partir de telles opérations primitives. La programmation impérative pouvant être considérée comme un cas particulier de programmation objet, nous choisissons d'étudier cette dernière catégorie de langages de programmation. Cette étude sera effectuée en essayant d'aborder les points suivants :

– Tout d'abord, nous voulons effectuer une analyse statique des programmes en utilisant une adaptation des sup-interprétations qui permette d'obtenir des certificats garantissant l'espace mémoire nécessaire à la bonne exécution d'un programme.
– Ensuite, nous cherchons à caractériser les ensembles des fonctions calculables en temps polynomial et en espace polynomial à l'aide du langage de programmation orienté-objet considéré.
– Enfin, nous cherchons à garantir la viabilité d'une telle étude en obtenant des heuristiques permettant de calculer les sup-interprétations d'un programme donné.

Nous allons essayer de répondre à toutes ces questions en élargissant la notion de sup-interprétation à un fragment de langage de programmation orienté objet incluant l'usage de méthodes non récursives. Notre langage est très proche des langages formels orientés objet étudiés dans [IPW01, AC96]. Cependant, puisque nous considérons des assignations, il est plus proche dans l'esprit d'un fragment du langage de programmation Java dont la sémantique opérationnelle est étudiée dans [DE99]. L'adjonction d'assignations dans un tel langage, c'est-à-dire la possibilité d'écrire dans des variables globales entraine irrémédiablement des effets de bords et rend l'étude de la complexité des programme plus ténue. C'est pour cette raison que nous chercherons au maximum à simplifier la syntaxe de notre langage. Dans l'ensemble, la syntaxe de notre langage est une restriction de la syntaxe du langage proposé dans [DE99] auquel nous ajoutons les constructions impératives while et loop.

Cette étude est grandement inspirée des récents travaux sur la complexité des programmes impératifs, et en particulier, des travaux de Niggl et Wunderlich [NW06] et de Jones et Kristiansen [KJ06]. Contrairement à ces travaux fondateurs, nous travaillons sur une algèbre de polynômes en lieu et place d'une algèbre de matrices. Un tel choix est dicté par au moins deux raisons. Premièrement, l'usage des polynômes donne une intuition plus claire sur les conditions de décroissances et de terminaison, et, permet d'éviter de nombreux aspects techniques et peu intuitifs sur les matrices. Deuxièmement, les polynômes permettent de traiter facilement les appels de méthode tandis que les matrices rendent leur considération difficile.

Dans ce chapitre, après avoir introduit la syntaxe et la sémantique de notre langage, nous allons définir la notion de sup-interprétation d'un programme orienté objet. Puis,

nous introduirons un nouveau critère, le critère fraternel, qui garantit que tout programme fraternel calcule des objets dont la taille est bornée polynomialement par la taille des objets en entrée, même lorsque le programme est défini avec des applications de méthodes. Nous étendrons ce critère aux méthodes de manière à obtenir des heuristiques pour synthétiser des sup-interprétations. Enfin, dans deux dernières sections, nous donnerons des caractérisations de l'ensemble des fonctions calculables en temps polynomial et de l'ensemble des fonctions calculables en espace polynomial à l'aide de programmes orientés-objet.

## III.1 Programmation orientée objet

### III.1.1 Syntaxe des programmes

Nous considérons des programmes orientés objet. Un programme se compose d'une séquence de classes, incluant une classe principale `main`, qui sont définies à l'aide d'identifiants de classe appartenant à l'ensemble `Class`. Dans la suite de ce chapitre, nous ne ferons plus de distinction entre une classe et son identifiant. Les identifiants de classe fournissent différents codages des données tels que les nombres unaires, en utilisant l'identifiant **S** pour une classe possédant un seul attribut et l'identifiant $\epsilon$ pour une classe sans aucun attribut, ou bien encore tels que les nombres binaires, en utilisant les identifiants de classe **1** et **0** à un attribut et l'identifiant $\epsilon$. Une classe $\mathtt{C} \in \mathtt{Class}$ se compose d'une séquence de déclarations d'attribut et d'une séquence de déclarations de méthode. La classe principale `main` se compose seulement de déclarations d'attribut et d'une commande, i.e. elle ne possède pas de déclaration de méthode. Un attribut $X$ est une variable globale appartenant à un ensemble fini *Var* et les déclarations d'attribut sont de la forme `var` $X$;. Une déclaration de méthode consiste en un identifiant de méthode `f` appartenant à l'ensemble *Fct*, une séquence d'arguments $x_1, \cdots, x_n$, appelés paramètres et appartenant à $\mathcal{P}$, et une commande `Cm`. Elle s'écrit sous la forme $\mathtt{f}(x_1, \cdots, x_n)\,\{\mathtt{Cm}\ ;\ \mathtt{return}\ X;\}$, où l'attribut $X$ correspond à l'objet retourné par la méthode. Afin d'éviter tout confusion entre les attributs, c'est-à-dire les variables globales, et les paramètres, les variables locales, nous utiliserons des lettres majuscules $X, Y, Z$ et respectivement des lettres minuscules $x, y, z$ pour les représenter. Une commande est soit une commande « skip », soit une assignation d'une valeur à un attribut $X := e$, soit une séquence de commandes $\mathtt{Cm}_1;\mathtt{Cm}_2$, soit une commande loop, soit une commande while, soit une conditionnelle. Une expression est soit un paramètre $x$, soit un attribut $X$, soit une application d'un opérateur $\mathbf{op}(e_1, \cdots, e_n)$, soit la création d'un nouvel objet en utilisant le constructeur `new` , soit une application de méthode de la forme $X.\mathtt{f}(e_1, \cdots, e_n)$ avec $\mathtt{f} \in \mathit{Fct}$. L'en-

semble des symboles d'opérateur **op** $\in Op$ rassemble toutes les opérations primitives du langage. La syntaxe précise du langage est résumée par la grammaire de la figure III.1.1.

$$
\begin{array}{lll}
\texttt{Attributs} & \ni A & ::= \texttt{var}\ X; \mid \texttt{var}\ X; A \\
\texttt{Expressions} & \ni e & ::= x \mid X \mid \mathbf{op}(e_1, \cdots, e_n) \mid X.\texttt{f}(e_1, \cdots, e_n) \mid \texttt{new C}(e_1, \cdots, e_n) \\
\texttt{Commandes} & \ni \texttt{Cm} & ::= \texttt{skip} \mid X := e \mid \texttt{Cm}_1; \texttt{Cm}_2 \\
 & & \quad \mid \texttt{if}\ e\ \texttt{then}\ \texttt{Cm}_1\ \texttt{else}\ \texttt{Cm}_2 \mid \texttt{loop}\ X\,\{\texttt{Cm}\} \mid \texttt{while}\ e\,\{\texttt{Cm}\} \\
\texttt{Méthodes} & \ni \texttt{M} & ::= \texttt{f}(x_1, \cdots, x_n)\,\{\texttt{Cm}\ ;\ \texttt{return}\ X;\} \\
\texttt{Classes} & \ni \texttt{C} & ::= \texttt{Class C}\,\{A\ ;\ M_1; \ldots; M_n;\} \\
 & \texttt{main} & ::= \texttt{Class main}\,\{A\ ;\ \texttt{Cm}\}
\end{array}
$$

FIG. III.1: Syntaxe des classes

Comme dans les chapitres I et II, nous utiliserons la notation $\overline{e}$ pour représenter une séquence d'expressions $e_1, \cdots, e_n$, lorsque $n$ est explicitement déterminé par le contexte. À présent, nous ajoutons les restrictions syntaxiques suivantes à notre langage :

- Tous les attributs apparaissant dans les méthodes d'une classe de `C` doivent appartenir aux attributs de cette classe. Tous les paramètres apparaissant dans une commande `Cm` d'une méthode $\texttt{f}(x_1, \cdots, x_n)\,\{\texttt{Cm}\ ;\ \texttt{return}\ X;\}$ doivent appartenir à la liste de paramètres $x_1, \cdots, x_n$.
- On suppose que les attributs et les méthodes de deux classes distinctes sont distincts deux à deux.
- Pour toute expression $e$ d'un programme, on suppose que les symboles de fonction $\texttt{f} \in Fct$ apparaissent uniquement à la racine d'une expression $e$. Cette restriction permet de traiter les effets de bord plus simplement. Elle n'est pas très restrictive puisque tout programme peut être transformé efficacement en un programme équivalent qui la satisfait, en ajoutant des nouveaux attributs pour les calculs intermédiaires.
- On définit un quasi-ordre $\geq_{Fct}$ sur les symboles de fonction de $Fct$. En effet, on pose $\texttt{f} \geq_{Fct} \texttt{g}$ si la méthode définissant `f` est de la forme $\texttt{f}(\overline{x})\,\{\texttt{Cm}\ ;\ \texttt{return}\ X;\}$ et qu'une assignation de la forme $Y := Z.\texttt{g}(\overline{e})$ apparait dans `Cm`. On définit la clôture réflexive et transitive de $\geq_{Fct}$, que nous noterons aussi $\geq_{Fct}$. $\texttt{f} >_{Fct} \texttt{g}$ si $\texttt{f} \geq_{Fct} \texttt{g}$ et $\texttt{g} \geq_{Fct} \texttt{f}$ n'est pas valide. Intuitivement, $\texttt{f} >_{Fct} \texttt{g}$ signifie que `g` ne peut pas appeler `f` pendant son exécution. On suppose que, pour tous symboles de fonction $\texttt{f}, \texttt{g} \in Fct$ tels que $\texttt{f} \geq_{Fct} \texttt{g}$, on ait $\texttt{f} >_{Fct} \texttt{g}$, i.e. il n'y a pas de récurrence dans le programme.

– Le programme ne peut pas écrire dans l'attribut $X$ pendant l'exécution d'une commande `loop` $X$ {`Cm`}. Ceci signifie qu'une assignation de la forme $X := e$ n'apparait ni dans la commande `Cm` ni dans aucune commande $\mathtt{Cm}_1$ d'une méthode de la forme $\mathtt{f}(\overline{x})\,\{\mathtt{Cm}_1\ ;\ \mathtt{return}\ Y;\}$ appliquée lors de l'évaluation de `Cm`.

**Exemple 37.** *Voici un exemple de programme de notre langage :*

```
Class Coord { var X; var Y;
              move(x,y) {X := add(X, x); Y := add(Y, y) ; return X; }
              getX() {skip ; return X; }
}
Class main { var W; var U; var V; Var Z;
             V := new Coord(W, U);
             Z := V.move(W, W);
             U := V.getX();
}
```

*où* `add` *est un opérateur qui calcule les additions unaire et binaire en fonction des objets qui lui sont donnés comme arguments. La classe* `Coord` *représente les coordonnées de position d'une personne. Elle possède deux méthodes* `move`*, qui permet de déplacer la personne, et* `getX`*, qui permet d'obtenir l'abscisse de la personne. Le programme principal commence par créer une nouvelle personne* $V := \mathtt{new\ Coord}(W, U);$ *puis la déplace* $Z := V.\mathtt{move}(W, W);$ *avant de calculer son abscisse* $U := V.\mathtt{getX}();$*.*

**Exemple 38.** *Voici un exemple de programme utilisant la commande* `loop` *et calculant la concaténation de deux listes :*

```
Class main { var X1; var X2; var Y;
             Y := X1;
             loop Y {X2 := new c(hd(X1), X2); X1 := tl(X1)}
   Class c { var H; var T; }
```

*Ce programme utilise l'identifiant de classe* **c** *pour coder les listes et deux opérateurs* **hd** *et* **tl** *qui calculent respectivement la tête et la queue d'une liste donnée en entrée.*

### III.1.2 Sémantique

Le domaine de calcul est l'ensemble des objets décrit dans [IPW01] et est défini inductivement par :

$$\texttt{Objects} \ni v ::= \texttt{new C}(v_1, \cdots, v_n) \quad \texttt{C} \in \texttt{Class}$$

Étant donné un programme ayant $n$ attributs $X_1, \cdots, X_n$, un objet $v_i$ est stocké dans chaque $X_i$ à tout instant à l'aide d'un magasin. Un magasin $\sigma$ consiste en une application des attributs de *Var* vers les objets de `Objects`. La notion de magasin correspond à celle de substitution des chapitre précédents. Étant donné un magasin $\sigma$ et un attribut $X$, la notation $\sigma\{X \leftarrow u\}$ signifie que l'objet stocké dans $X$ est remplacé par l'objet $u$ dans $\sigma$. Étant donnés une expression $e$ et un magasin $\sigma$, on utilise la notation $\langle e, \sigma\rangle \downarrow \langle u, \sigma'\rangle$ lorsque l'expression $e$ s'évalue en $u$ et lorsque le magasin $\sigma$ est remplacé par le magasin $\sigma'$ pendant cette évaluation. Étant donnée une commande `Cm`, on utilise la notation $\langle \texttt{Cm}, \sigma\rangle \downarrow \langle\sigma'\rangle$, si $\sigma$ est remplacé par $\sigma'$ durant l'exécution de `Cm`. Étant donné un programme **p** de classe principale `Class main` $\{A; \texttt{Cm}\}$ et un magasin $\sigma$, **p** calcule un magasin $\sigma'$ défini par $\langle \texttt{Cm}, \sigma\rangle \downarrow \langle\sigma'\rangle$.

$$\frac{}{\langle X, \sigma\rangle \downarrow \langle X\sigma, \sigma\rangle} \; X \in \mathit{Var}$$

$$\frac{\forall i \in \{1, \dots, n\} \;\; \langle e_i, \sigma\rangle \downarrow \langle u_i, \sigma\rangle}{\langle \mathbf{op}(e_1, \cdots, e_n), \sigma\rangle \downarrow \langle [\![\mathbf{op}]\!](u_1, \cdots, u_n), \sigma\rangle} \; \mathbf{op} \in \mathit{Op}$$

$$\frac{\begin{matrix} X\sigma = \texttt{new C}(\overline{v}) & \text{Class } \texttt{C}\,\{\dots \texttt{var } \overline{V}; \dots \texttt{f}(\overline{x})\,\{\texttt{Cm} \; ; \; \texttt{return } Y;\}\,;\dots\} \\ \forall i\,\langle e_i, \sigma\rangle \downarrow \langle u_i, \sigma\rangle & \left\langle \texttt{Cm}[\overline{x}/\overline{u}], \sigma\left\{\overline{V} \leftarrow \overline{v}\right\}\right\rangle \downarrow \langle\sigma'\rangle \end{matrix}}{\langle X.\texttt{f}(\overline{e}), \sigma\rangle \downarrow \langle Y\sigma', \sigma'\rangle} \; \texttt{f} \in \mathit{Fct}$$

$$\frac{\forall i \in \{1, \dots, n\} \;\; \langle e_i, \sigma\rangle \downarrow \langle u_i, \sigma\rangle}{\langle \texttt{new C}(e_1, \cdots, e_n), \sigma\rangle \downarrow \langle \texttt{C}(u_1, \cdots, u_n), \sigma\rangle} \; \texttt{C} \in \texttt{Class}$$

FIG. III.2: Sémantique opérationnelle des expressions

La sémantique opérationnelle de notre langage est définie en figures III.2 et III.3 et s'inspire de la sémantique opérationnelle du fragment de Java étudié dans [DE99]. Elle est plus proche de la sémantique de [DE99] que de celle de [IPW01] car nous utilisons des assignations de la forme $X := e$ et, par conséquent, des effets de bord sont présents

$$\frac{\langle e,\sigma\rangle \downarrow \langle u,\sigma'\rangle}{\langle X := e\rangle \downarrow \langle \sigma'\{X \leftarrow u\}\rangle}$$

$$\frac{}{\langle \texttt{skip},\sigma\rangle \downarrow \langle \sigma\rangle}$$

$$\frac{\langle \texttt{Cm}_1,\sigma\rangle \downarrow \langle \sigma'\rangle \quad \langle \texttt{Cm}_2,\sigma'\rangle \downarrow \langle \sigma''\rangle}{\langle \texttt{Cm}_1;\texttt{Cm}_2,\sigma\rangle \downarrow \langle \sigma''\rangle}$$

$$\frac{\langle e,\sigma\rangle \downarrow \langle \texttt{new }\mathbf{1}(),\sigma\rangle, \langle \texttt{new }\mathbf{0}(),\sigma\rangle \;\; \texttt{ou} \;\; \langle \texttt{new }\mathbf{c}(\ldots),\sigma\rangle}{\langle \texttt{if } e \texttt{ then Cm}_1 \texttt{ else Cm}_2,\sigma\rangle \downarrow \langle \texttt{Cm}_1,\sigma\rangle, \langle \texttt{Cm}_2,\sigma\rangle \;\; \texttt{ou} \;\; \langle \texttt{skip},\sigma\rangle} \texttt{avec } \mathbf{c} \neq \mathbf{1},\mathbf{0}$$

$$\frac{\langle X_i,\sigma\rangle \downarrow \langle v_i,\sigma\rangle}{\langle \texttt{loop } X_i\,\{\texttt{Cm}\},\sigma\rangle \downarrow \left\langle \texttt{Cm}^{|v_i|},\sigma\right\rangle} \texttt{avec } \texttt{Cm}^n = \texttt{Cm};\texttt{Cm}^{n-1} \texttt{ et } \texttt{Cm}^0 = \texttt{skip}$$

$$\frac{\langle e,\sigma\rangle \downarrow \langle \texttt{new }\mathbf{1}(),\sigma\rangle \;\; \texttt{ou} \;\; \langle u,\sigma\rangle}{\langle \texttt{while } e\,\{\texttt{Cm}\},\sigma\rangle \downarrow \langle \texttt{Cm};\texttt{while } X\,\{\texttt{Cm}\},\sigma\rangle \;\; \texttt{ou} \;\; \langle \texttt{skip},\sigma\rangle} \texttt{si } u \neq \texttt{new }\mathbf{1}()$$

Fig. III.3: Sémantique opérationnelle des commandes

lors de l'exécution de nos programmes. Notre sémantique opérationnelle n'utilise pas de références puisqu'elles sont implicites en Java et que nous ne cherchons pas à contrôler les buffer-overflow dus à un éventuel manque de mémoire dans le tas. Pour une analyse statique de tels problèmes utilisant les interprétations abstraites [CC77], nous invitons le lecteur à lire les papiers de [Min06, CCF+05]. Un opérateur **op** d'arité $n$ est évalué sans aucun effet de bord et calcule une fonction partielle $[\![\mathbf{op}]\!]$ de $\texttt{Objects}^n$ dans `Objects`.

Si `f` est définie par une méthode de la forme $\texttt{f}(\overline{x})\,\{\texttt{Cm ; return } Y;\}$ alors, étant donné un magasin $\sigma$, l'évaluation de $X.\texttt{f}(\overline{u})$ est effectuée en commençant par évaluer son corps $\texttt{Cm}[\overline{x}/\overline{u}]$, où $[\overline{x}/\overline{u}]$ dénote la substitution de chaque paramètre $x_j$ par l'objet $u_j$, dans le contexte d'un objet $X\sigma$, puis, en retournant la valeur stockée dans l'attribut $Y$ correspondant à l'objet $X\sigma$. Lors d'une application de méthode $X.\texttt{f}(\overline{u})$ les expressions $\overline{u}$ se comportent comme des paramètres passés par valeur.

La commande `skip` ne fait rien. La commande $X := e$ assigne l'objet calculé par $e$ à l'attribut $X$. La commande $X := \texttt{new C}(e_1, \cdots, e_n)$ commence par évaluer les expressions $e_1, \cdots, e_n$ en les objets $v_1, \cdots, v_n$, puis, crée un nouvel objet $\texttt{new C}(v_1, \cdots, v_n)$ qu'elle assigne à l'attribut $X$ dans le magasin $\sigma$. L'exécution de $\texttt{Cm}_1;\texttt{Cm}_2$ correspond à l'exécution séquentielle des commandes $\texttt{Cm}_1$ et $\texttt{Cm}_2$. La conditionnelle $\texttt{if } b \texttt{ then Cm}_1 \texttt{ else Cm}_2$ exécute la commande $\texttt{Cm}_1$, la commande $\texttt{Cm}_2$ ou la commande `skip` suivant que l'expression

$b$ soit respectivement évaluée en les ojets $\texttt{new}\ \mathbf{1}()$, $\texttt{new}\ \mathbf{0}()$ ou en tout autre objet.

**Définition III.1.** *La taille $|v|$ d'un objet $v$ est définie comme le nombre d'identifiants de classe ayant un nombre strictement positif d'attributs dans $v$, i.e. $|\texttt{new C}(v_1, \cdots, v_n)| = \sum_{j=1}^{n} |v_j| + 1$ et $|\texttt{new C}()| = 0$.*

La commande $\texttt{loop}\ X\,\{\texttt{Cm}\}$ exécute $|v|$ fois la commande $\texttt{Cm}$ si $v$ est l'objet stocké dans $X$, i.e. $X\sigma = v$. Finalement, la commande $\texttt{while}\ b\,\{\texttt{Cm}\}$ est évaluée à $\texttt{Cm};\texttt{while}\ b\,\{\texttt{Cm}\}\,;$, si l'évaluation de $b$ est égale à l'objet $\texttt{new}\ \mathbf{1}()$, et est évaluée à $\texttt{skip}$ dans tous les autres cas.

**Exemple 39.** *Considérons le programme de l'exemple 37. Pour tout magasin $\sigma$ tel que $U\sigma = u, V\sigma = v, W\sigma = w$ et $Z\sigma = z$, on a :*

$$\langle \texttt{new Coord}(W, U), \sigma\rangle \downarrow \langle \texttt{new Coord}(w, u), \sigma\rangle$$
$$\langle V := \texttt{new Coord}(W, U), \sigma\rangle \downarrow \langle \sigma\,\{V \leftarrow \texttt{new Coord}(w, u)\}\rangle$$

*De plus, si $V\sigma = \texttt{new Coord}(w, u)$ alors :*

$$\langle V.\texttt{getX}(), \sigma\rangle \downarrow \langle w, \sigma\rangle$$
$$\langle U := V.\texttt{getX}(), \sigma\rangle \downarrow \langle \sigma\,\{U \leftarrow w\}\rangle$$

**Exemple 40.** *Considérons le programme de l'exemple 38. Pour tout magasin $\sigma$ tel que $X_1\sigma = \texttt{new}\ \mathbf{c}(\texttt{new}\ \mathbf{True}(), \texttt{new}\ \mathbf{c}(\texttt{new}\ \mathbf{True}(), \texttt{new}\ \mathbf{False}()))$ et $X_2\sigma = \texttt{new}\ \mathbf{c}(\texttt{new}\ \mathbf{False}(), \texttt{new}\ \mathbf{False}())$, et en omettant le symbole $\texttt{new}$ pour faciliter la lecture, on a :*

$$\langle Y := X_1, \sigma\rangle \downarrow \langle \sigma\,\{Y \leftarrow \mathbf{c}(\mathbf{True}(), \mathbf{c}(\mathbf{True}(), \mathbf{False}()))\}\rangle$$
$$\langle \texttt{loop}\ Y\,\{X_2 := \mathbf{c}(\mathbf{hd}(X_1), X_2; X_1 := \mathbf{tl}(X_1)\}\,, \sigma\rangle \downarrow \langle \sigma\,\{X_1 \leftarrow \mathbf{False}(), X_2 \leftarrow \mathbf{c}(\mathbf{False}(), \mathbf{c}(\mathbf{True}(), \mathbf{c}(\mathbf{True}(), \mathbf{c}(\mathbf{False}(), \mathbf{False}())))))\}\rangle$$

## III.2 Sup-interprétations et poids

### III.2.1 Assignations

**Définition III.2.** *Étant donné l'ensemble des opérateurs Op, l'assignation $I$ de Op est une fonction totale qui associe une fonction $I(\mathbf{op}) : (\mathbb{R}^+)^m \longmapsto \mathbb{R}^+$ à chaque symbole d'opérateur d'arité $m$.*

*Étant donné un symbole de fonction $\texttt{f} \in Fct$ d'arité $m$, l'assignation de $\texttt{f}$, notée $I(\texttt{f})$, est une fonction de $(\mathbb{R}^+)^{m+1}$ dans $\mathbb{R}^+$.*

*Étant donné un identifiant de classe* $\mathtt{C}$ *correspondant à une classe possédant* $n$ *attributs, l'assignation de* $\mathtt{C}$*, notée* $I(\mathtt{C})$*, est une fonction de* $(\mathbb{R}^+)^n$ *dans* $\mathbb{R}^+$*.*

*Étant donnée une classe* $\mathtt{C}$*, une assignation* $I$ *de la classe* $\mathtt{C}$ *est l'union des assignations de symboles de fonction appartenant à un domaine* $dom(I) \subseteq Fct$*, correspondant à un sous-ensemble des méthodes de la classe* $\mathtt{C}$*, et de l'assignation de l'identifiant de classe* $\mathtt{C}$*.*

*Étant donné un programme* $\mathbf{p}$*, l'assignation* $I$ *de* $\mathbf{p}$ *est l'union des assignations de chaque classe* $\mathtt{C}$ *de* $\mathtt{Class}$ *et de l'assignation de l'ensemble* $Op$*.*

*L'assignation d'un programme* $I$ *est définie sur une expression* $e$ *si chaque symbole de* $Op \cup Fct \cup \mathtt{Class}$ *apparaissant dans* $e$ *appartient à* $dom(I)$*. Supposons que l'assignation* $I$ *soit définie sur une expression* $e$*, l'assignation partielle de* $e$ *relativement à* $I$*, que nous notons* $I^*(e)$ *est l'extension canonique de l'assignation* $I$ *définie comme suit :*

1. *Si* $\diamond$ *est dans* $Var \cup \mathcal{P}$*, alors* $I^*(\diamond) = \square$*, avec* $\square$ *une nouvelle variable à valeur dans* $\mathbb{R}^+$*,*
2. *Si* $\mathbf{op}$ *est un symbole d'opérateur de* $Op$ *d'arité* $m$ *et si les* $e_1, \cdots, e_m$ *sont des expressions, alors, on a :*
$$I^*(\mathbf{op}(e_1, \cdots, e_m)) = I(\mathbf{op})(I^*(e_1), \ldots, I^*(e_m))$$
3. *Si* $\mathtt{C}$ *est un identifiant de classe de* $\mathtt{Class}$ *ayant* $m$ *attributs et si les* $e_1, \cdots, e_m$ *sont des expressions, alors, on a :*
$$I^*(\mathtt{new\ C}(e_1, \cdots, e_m)) = I(\mathtt{C})(I^*(e_1), \ldots, I^*(e_m))$$
4. *Si* $\mathtt{f} \in Fct$ *est un symbole d'arité* $m$ *et si les* $e_1, \cdots, e_m$ *sont des expressions, alors, on a :*
$$I^*(X.\mathtt{f}(e_1, \cdots, e_m)) = I(\mathtt{f})(I^*(e_1), \ldots, I^*(e_m), I^*(X))$$

*Tout comme dans les chapitres I et II, si* $\overline{e}$ *est une séquence d'expressions* $e_1, \cdots, e_k$*, on utilisera parfois la notation* $I^*(\overline{e}) = I^*(e_1), \ldots, I^*(e_k)$*.*

Commençons par remarquer que l'assignation $I^*(e)$ d'une expression $e$ ayant $m$ paramètres $x_1, \cdots, x_m$ et apparaissant dans une classe $\mathtt{C}$ ayant $n$ attributs dénote une fonction de $(\mathbb{R}^+)^{n+m} \to \mathbb{R}^+$. Nous utiliserons donc la notation $I^*(e)(a_1, \cdots, a_n, b_1, \cdots, b_m)$ pour signifier que chaque $I^*(X_i)$ est remplacé par la valeur $a_i \in \mathbb{R}^+$ et que chaque $I^*(x_j)$ est remplacé par la valeur $b_j \in \mathbb{R}^+$ dans $I^*(e)$.

À présent, la notion d'assignation additive change légèrement puisque nous avons changé de domaine de calcul. Cependant, si l'on considère que tout identifiant de classe

est un symbole de constructeur alors cette notion est similaire à celles des chapitres I et II.

**Définition III.3.** *L'assignation d'un identifiant de classe* $\mathtt{C} \in \mathtt{Class}$ *ayant* $m$ *attributs est* additive *si :*

$$I(\mathtt{C})(\diamond_1, \cdots, \diamond_m) = \sum_{i=1}^{m} \diamond_i + \alpha_{\mathtt{C}} \text{ où } \alpha_{\mathtt{C}} \geq 1 \qquad \text{lorsque } m > 0$$

$$I(\mathtt{C}) = 0 \qquad \text{sinon.}$$

*Si l'assignation de chaque identifiant de classe possédant un nombre strictement positif d'attributs est additive alors l'assignation du programme est additive.*

**Lemme III.4.** *Étant donné un programme* $\mathbf{p}$ *admettant une assignation additive* $I$*, il existe une constante* $\alpha$ *telle que pour tout objet* $v \in \mathtt{Objects}$*, les inégalités suivantes soient satisfaites :*

$$|v| \leq I^*(v) \leq \alpha \times |v|$$

*Démonstration.* Ce résultat est une conséquence directe du lemme I.12. Il suffit d'adapter la preuve aux objets. □

### III.2.2 Sup-interprétations

**Définition III.5.** *Étant donné un programme* $\mathbf{p}$*, une sup-interprétation est une assignation* $\theta$ *de* $\mathbf{p}$ *qui vérifie :*

1. *L'assignation* $\theta$ *est monotone. C'est à dire, pour tout symbole* $b \in dom(\theta)$*, la fonction* $\theta(b)$ *satisfait :*

$$\forall i,\ \diamond_i \geq \diamond'_i \Rightarrow \theta(b)(\ldots, \diamond_i, \ldots) \geq \theta(b)(\ldots, \diamond'_i, \ldots)$$

2. *Pour tout objet* $v \in \mathtt{Object}$, $\theta^*(v) \geq |v|$
3. – *Pour tout symbole d'opérateur* $\mathbf{op}$*, pour toute séquence d'objets* $\overline{v}$*, et pour tout magasin* $\sigma$ *si* $\langle \mathbf{op}(\overline{v}), \sigma \rangle \to \langle u, \sigma \rangle$ *alors :*

$$\theta(\mathbf{op})(\theta^*(\overline{v})) \geq \theta^*(u)$$

   – *Pour tout symbole de fonction* $\mathtt{f} \in dom(\theta)$ *d'arité* $m$*, pour toute séquence d'objets* $\overline{v} \in \mathtt{Objects}$ *et pour tout magasin* $\sigma$ *si* $\langle X_i.\mathtt{f}(\overline{v}), \sigma \rangle \downarrow \langle u, \sigma' \rangle$ *alors :*

$$\theta(\mathtt{f})(\theta^*(\overline{v}), \theta^*(X_i\sigma)) \geq \max(\theta^*(u), \theta^*(X_i\sigma'))$$

**Exemple 41.** *Supposons que l'opérateur* `add` *de l'exemple 37 soit défini sur une classe d'entiers unaires utilisant les identifiants de classe* $\mathbf{S}$ *et* $\epsilon$ *d'arités respectives* 1 *et* 0*. Le programme admet la sup-interprétation polynomiale et additive suivante :*

$$\begin{aligned}\theta(\texttt{move})(x,y,\diamond) &= x+y+\diamond\\ \theta(\texttt{getX})(\diamond) &= \diamond\\ \theta(\texttt{add})(\diamond_1,\diamond_2) &= \theta(\texttt{Coord})(\diamond_1,\diamond_2) = \diamond_1+\diamond_2+1\\ \theta(\texttt{new}\ \ \mathbf{S})(\diamond) &= \diamond+1\\ \theta(\texttt{new}\ \ \epsilon) &= 0\end{aligned}$$

*En effet, ces fonctions sont toutes monotones. Pour tout entier unaire* $(\texttt{new}\ \ \mathbf{S})^v(\epsilon)$*, nous laissons au lecteur le soin de vérifier que* $\theta^*((\texttt{new}\ \ \mathbf{S})^v(\epsilon)) = |(\texttt{new}\ \ \mathbf{S})^v(\epsilon)| = v$*. De plus, la condition 3 de la définition III.5 est elle-aussi vérifiée. En particulier, en utilisant les notations* $\mathbf{S} = \texttt{new}\ \ \mathbf{S}$*,* $\epsilon = \texttt{new}\ \ \epsilon()$*,* $\texttt{Coord} = \texttt{new}\ \ \texttt{Coord}$*,* $\mathbf{S}^{n+1}(\epsilon) = \mathbf{S}(\mathbf{S}^n(\epsilon))$ *et* $\mathbf{S}^0(\epsilon) = \epsilon$*, pour tout tout magasin* $\sigma$*, si* $\langle Y.\texttt{getX}(),\sigma\rangle \downarrow \langle w,\sigma\rangle$ *alors :*

$$\begin{aligned}\theta^*(\texttt{getX})(\theta^*(Y\sigma)) &= \theta^*(Y\sigma) && \textit{Par définition } \theta(\texttt{getX})\\ &= \theta^*(\texttt{Coord}(w,z)) && \textit{Par définition de de } \texttt{getX}\\ &= \theta^*(w)+\theta^*(z)+1 && \textit{Par définition de } \theta(\texttt{Coord})\\ &\geq \max(\theta^*(w),\theta^*(Y\sigma))\end{aligned}$$

**Exemple 42.** *Considérons le programme de l'exemple 38. Ce programme admet la sup-interprétation polynomiale et additive suivante* $\theta(\mathbf{c})(\diamond_1,\diamond_2) = \diamond_1+\diamond_2+1$*,* $\theta(\mathbf{hd})(\diamond) = \theta(\mathbf{tl})(\diamond) = \diamond$ *et* $\theta(\mathbf{True}) = \theta(\mathbf{False}) = 0$*. En effet, on vérifie aisément les conditions 1 et 2 de la définition III.5. De plus, on a pour tout tout magasin* $\sigma$ *si* $\langle \mathbf{hd}(\texttt{new}\ \ \mathbf{c}(u_1,u_2)),\sigma\rangle \downarrow \langle u_1,\sigma\rangle$*, alors :*

$$\begin{aligned}\theta^*(\mathbf{hd})(\theta^*(\texttt{new}\ \ \mathbf{c}(u_1,u_2))) &= \theta^*(\texttt{new}\ \ \mathbf{c}(u_1,u_2)) && \textit{Par définition } \theta(\mathbf{hd})\\ &= \theta^*(u_1)+\theta^*(u_2)+1 && \textit{Par définition de } \theta(\mathbf{c})\\ &\geq \theta^*(u_1)\end{aligned}$$

**Lemme III.6.** *Étant donné un programme* $\mathbf{p}$ *admettant une sup-interprétation* $\theta$ *définie sur une expression* $e$ *ayant* $m$ *paramètres* $\overline{x}$ *et apparaissant dans une classe* `C` *possédant* $n$ *attributs* $X_1,\cdots,X_n$*, alors, pour toute séquence de* $m$ *objets* $\overline{u}$*,* $\theta^*(e[\overline{x}/\overline{u}])$ *dénote une fonction de* $(\mathbb{R}^+)^n$ *dans* $\mathbb{R}^+$ *qui satisfait :*

*Pour tout magasin* $\sigma$*, si* $\langle e[\overline{x}/\overline{u}],\sigma\rangle \downarrow \langle v,\sigma'\rangle$ *alors*

$$\theta^*(e[\overline{x}/\overline{u}])(\theta^*(X_1\sigma),\ldots,\theta^*(X_n\sigma)) \geq \theta^*(v)$$

*De plus, si* $e = X_i.\mathtt{f}(e_1, \cdots, e_n)$*, on a :*

$$\theta^*(e[\overline{x}/\overline{u}])(\theta^*(X_1\sigma), \ldots, \theta^*(X_n\sigma)) \geq \theta^*(X_i\sigma')$$

*Démonstration.* Supposons que la classe $\mathtt{C}$ possède $n$ attributs et une sup-interprétation $\theta$. Nous allons démontrer ce lemme par induction structurelle sur une expression $e$ telle que $\langle e[\overline{x}/\overline{u}], \sigma\rangle \downarrow \langle v, \sigma'\rangle$. Supposons qu'un symbole de fonction apparaisse dans $e$, i.e. $e = X_j.\mathtt{f}(e_1, \cdots, e_m)$ avec $\mathtt{f} \in \mathit{Fct}$. Les restrictions syntactiques de la section III.1 impliquent qu'aucun symbole de fonction n'apparait dans les $e_1, \cdots, e_m$. Par conséquent, il n'y a aucun effet de bord pendant l'évaluation des arguments de la fonction. En d'autres termes, $\langle e_i[\overline{x}/\overline{u}], \sigma\rangle \downarrow \langle u_i, \sigma\rangle$, pour un objet $u_i$ donné.

En utilisant la notation $\overline{X}\sigma = X_1\sigma, \ldots, X_n\sigma$, supposons, par Hypothèse d'Induction, que nous ayons :

$$\theta^*(e_i[\overline{x}/\overline{u}])(\theta^*(\overline{X}\sigma)) \geq \theta^*(u_i)$$

Par conséquent :

| | |
|---|---|
| $\theta^*(e[\overline{x}/\overline{u}])(\theta^*(\overline{X}\sigma))$ | |
| $= \theta(\mathtt{f})(\theta^*(e_1[\overline{x}/\overline{u}]), \ldots, \theta^*(e_m[\overline{x}/\overline{u}]), \theta^*(X_i))(\theta^*(\overline{X}\sigma))$ | Par Définition |
| $= \theta(\mathtt{f})(\theta^*(e_1[\overline{x}/\overline{u}])(\theta^*(\overline{X}\sigma)), \ldots, \theta^*(e_m[\overline{x}/\overline{u}])(\theta^*(\overline{X}\sigma)), \theta^*(X_i\sigma))$ | Par Application |
| $\geq \theta(\mathtt{f})(u_1\sigma, \cdots, u_m\sigma, \theta^*(X_i\sigma))$ | Par H.I. |
| $\geq \max(\theta^*(v), \theta^*(X_i\sigma'))$ | Définition III.5 |

Les cas de figure où $e = \mathtt{new\ C}(\overline{e})$ ou bien $e = \mathbf{op}(\overline{e})$ se démontrent de manière identique. □

**Exemple 43.** *Considérons l'expression* $V.\mathtt{move}(W, W)$ *du programme de l'exemple 37. Cette expression ne fait intervenir aucun paramètre. Comme le programme admet la sup-interprétation additive et polynomiale suivante* $\theta(\mathtt{add})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2$*,* $\theta(\mathbf{S})(\diamond) = \diamond + 1$ *et* $\theta(\epsilon) = 0$*,* $\theta(\mathtt{move})(\diamond_1, \diamond_2, \diamond_3) = \diamond_1 + \diamond_2 + \diamond_3$ *et* $\theta(\mathtt{Coord})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2 + 1$*, en utilisant les notations* $\mathbf{S} = \mathtt{new\ \ S}$*,* $\epsilon = \mathtt{new}\ \ \epsilon()$ *et* $\mathtt{Coord} = \mathtt{new\ \ Coord}$ *et pour tout magasin* $\sigma$ *satisfaisant :*

$$U\sigma = u, V\sigma = \mathtt{new\ \ Coord}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon)), W\sigma = \mathbf{S}^w(\epsilon), Z\sigma = z$$

*ainsi que :*

$$\langle V.\mathtt{move}(W, W), \sigma\rangle \downarrow \left\langle \mathbf{S}^{u_1+w}(\epsilon), \sigma\left\{V \leftarrow \mathtt{Coord}(\mathbf{S}^{u_1+w}(\epsilon), \mathbf{S}^{u_2+w}(\epsilon))\right\}\right\rangle$$

*on a :*

$$\begin{aligned}
&\theta^*(V.\mathtt{move}(W\sigma, W\sigma))(\theta^*(u), \theta^*(\mathtt{Coord}(\mathbf{S}^{u_1}(\epsilon)), \theta^*(\mathbf{S}^{u_2}(\epsilon)), \theta^*(\mathbf{S}^{w}(\epsilon), z))) \\
&= \theta(\mathtt{move})(\theta^*(W\sigma), \theta^*(W\sigma), \theta^*(\mathtt{Coord}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon)))) \\
&= \theta^*(W\sigma) + \theta^*(W\sigma) + \theta^*(\mathtt{Coord}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon))) \\
&= \theta^*(\mathbf{S}^{w}(\epsilon)) + \theta^*(\mathbf{S}^{w}(\epsilon)) + \theta^*(\mathbf{S}^{u_1}(\epsilon)) + \theta^*(\mathbf{S}^{u_2}(\epsilon)) + 1 \\
&= 2 \times w + u_1 + u_2 + 1 \\
&\geq \max(w + u_1, 2 \times w + u_1 + u_2 + 1) \\
&= \max(\theta^*(\mathbf{S}^{u_1+w}(\epsilon)), \theta^*(\mathtt{Coord}(\mathbf{S}^{u_1+w}(\epsilon), \mathbf{S}^{u_2+w}(\epsilon))))
\end{aligned}$$

### III.2.3 Poids

La notion de poids permet le contrôle de la taille des objets stockés par les attributs durant l'exécution d'une commande loop ou d'une commande while. Une poids consiste simplement en une fonction partielle à valeur sur les commandes. Les poids vont dépendre fortement de la forme des commandes considérées de sorte qu'il est nécessaire d'effectuer une distinction entre les commandes. Dans cette optique, on définit la relation d'ordre partiel $\sqsubseteq$ sur les commandes par $\mathtt{Cm}_1 \sqsubseteq \mathtt{Cm}$ si :

- $\mathtt{Cm} = \mathtt{Cm}_1$,
- $\mathtt{Cm} = \mathtt{loop}\ X\{\dots;\mathtt{Cm}_2;\dots\}$ et $\mathtt{Cm}_1 \sqsubseteq \mathtt{Cm}_2$,
- ou $\mathtt{Cm} = \mathtt{while}\ e\{\dots;\mathtt{Cm}_2;\dots\}$ et $\mathtt{Cm}_1 \sqsubseteq \mathtt{Cm}_2$.

et sa clôture réflexive et transitive, que nous noterons aussi $\sqsubseteq$, ainsi que la relation d'ordre stricte, notée $\sqsubset$, définie par $\mathtt{Cm} \sqsubset \mathtt{Cm}_1$ si et seulement si $\mathtt{Cm} \sqsubseteq \mathtt{Cm}_1$ et $\mathtt{Cm} \neq \mathtt{Cm}_1$.

**Définition III.7.** *Une commande* $\mathtt{Cm}$ *est dite :*

- *externe s'il n'existe pas de commande* $\mathtt{Cm}_1$ *telle que* $\mathtt{Cm} \sqsubset \mathtt{Cm}_1$*. Une commande externe correspond à une boucle loop ou while non imbriquée.*
- *minimum s'il n'y a pas de commandes* $\mathtt{Cm}_1$ *et* $\mathtt{Cm}_2$ *et d'expression* $e$ *telles que* $\mathtt{Cm} = \mathtt{Cm}_1;\mathtt{Cm}_2$ *ou* $\mathtt{Cm} = \mathtt{if}\ e\ \mathtt{then}\ \mathtt{Cm}_1\ \mathtt{else}\ \mathtt{Cm}_2$*.*
- *de type while si elle est externe et minimum et s'il existe une commande* $\mathtt{Cm}_1 = \mathtt{while}\ e\{\mathtt{Cm}_2\}$ *telle que* $\mathtt{Cm}_1 \sqsubseteq \mathtt{Cm}$*.*
- *de type loop si elle est externe et minimum, s'il existe une commande* $\mathtt{Cm}_1$ *de la forme* $\mathtt{loop}\ X\{\mathtt{Cm}_2\}$ *telle que* $\mathtt{Cm}_1 \sqsubseteq \mathtt{Cm}$ *et si elle n'est pas de type while.*

**Exemple 44.** *Nous illustrons les différentes notions introduites ci-dessus par l'exemple*

*suivant :*

$$
\begin{array}{l}
\texttt{Class main}\,\{ \\
\qquad \texttt{Cm}_1 : \texttt{loop}\ X\,\{ \\
\qquad\qquad \texttt{Cm}_2 : \texttt{while}(Y > 0)\,\{\ \texttt{Cm}_3 : Y = Y - 1;\ \} \\
\qquad\quad \}\,; \\
\qquad \texttt{Cm}_5 : \begin{cases} X := X + 1; \\ \texttt{Cm}_4 : \texttt{loop}\ \ X\,\{Y := Y + 1;\}\,; \end{cases} \\
\}
\end{array}
$$

*où les* $\texttt{Cm}_i$ *sont des labels utilisés afin de désigner les commandes.*

*La commande* $\texttt{Cm}_5$ *est externe mais n'est pas minimum. Les commandes* $\texttt{Cm}_1$ *et* $\texttt{Cm}_4$ *sont externes et minimums, et les commandes* $\texttt{Cm}_2$ *et* $\texttt{Cm}_3$ *ne sont pas externes puisqu'elles sont toutes contenues dans une commande loop. La commande* $\texttt{Cm}_1$ *est de type while, puisqu'elle contient une commande while. Finalement, seule la commande* $\texttt{Cm}_4$ *est de type loop.*

**Définition III.8.** *Étant donné un programme* $\mathbf{p}$ *possédant une classe principale de* $n$ *attributs, le poids d'une commande* $\omega$ *est une fonction partielle qui assigne à :*
- *chaque commande* $\texttt{Cm}$ *de type loop, une fonction totale* $\omega_{\texttt{Cm}}$ *de* $(\mathbb{R}^+)^{n+1}$ *dans* $\mathbb{R}^+$
- *chaque commande* $\texttt{Cm}$ *de type while, une fonction totale* $\omega_{\texttt{Cm}}$ *de* $(\mathbb{R}^+)^n$ *dans* $\mathbb{R}^+$

*qui satisfont :*

1. $\omega_{\texttt{Cm}}$ *est monotone* $\forall i,\ \diamond_i \geq \diamond'_i \Rightarrow \omega_{\texttt{Cm}}(\ldots, \diamond_i, \ldots) \geq \omega_{\texttt{Cm}}(\ldots, \diamond'_i, \ldots)$
2. $\omega_{\texttt{Cm}}$ *possède la propriété sous-terme* $\forall i,\ \forall \diamond_i \in \mathbb{R}^+\ \omega_{\texttt{Cm}}(\ldots, \diamond_i, \ldots) \geq \diamond_i$

*Un poids* $\omega$ *est polynomial si chaque* $\omega_{\texttt{Cm}}$ *est une fonction de* **Max-Poly** $\{\mathbb{R}^+\}$.

## III.3 Critère fraternel

### III.3.1 Définition

Le critère fraternel fournit des contraintes sur les poids et les sup-interprétations afin de borner la taille des objets calculés par le programme par un polynôme en la taille des entrées.

**Définition III.9.** *Un programme ayant une classe principale à* $n$ *attributs* $X_1, \cdots, X_n$ *est* fraternel *s'il existe une sup-interprétation polynomiale et additive* $\theta$ *et un poids polynomial* $\omega$ *tels que* $\theta(X_i) = \diamond_i$ *et :*

1. *Pour toute commande* `Cm` *de type loop de la classe principale, on a :*
   *Pour toute expression* $e = X_j.\mathtt{f}(e_1, \cdots, e_m)$ *apparaissant dans* `Cm` *:*

$$\omega_{\mathtt{Cm}}(T+1, \diamond_1, \cdots, \diamond_n) \geq \omega_{\mathtt{Cm}}(T, \diamond_1, \ldots, \diamond_{j-1}, \theta^*(e)(\overline{\diamond}), \diamond_{j+1}, \ldots, \diamond_n)$$

   *Pour toute commande* $X_i := e \sqsubseteq \mathtt{Cm}$*, on a :*

$$\omega_{\mathtt{Cm}}(T+1, \diamond_1, \cdots, \diamond_n) \geq \omega_{\mathtt{Cm}}(T, \diamond_1, \ldots, \diamond_{i-1}, \theta^*(e)(\overline{\diamond}), \diamond_{i+1}, \ldots, \diamond_n)$$

   *avec* $T$ *une nouvelle variable.*

2. *Pour toute commande* `Cm` *de type while de la classe principale, on a :*
   *Pour toute expression* $e = X_j.\mathtt{f}(e_1, \cdots, e_m)$ *apparaissant dans* `Cm` *:*

$$\omega_{\mathtt{Cm}}(\diamond_1, \cdots, \diamond_n) \geq \omega_{\mathtt{Cm}}(\diamond_1, \ldots, \diamond_{j-1}, \theta^*(e)(\overline{\diamond}), \diamond_{j+1}, \ldots, \diamond_n)$$

   *Pour toute commande* $X_i := e \sqsubseteq \mathtt{Cm}$*, on a :*

$$\omega_{\mathtt{Cm}}(\diamond_1, \cdots, \diamond_n) \geq \omega_{\mathtt{Cm}}(\diamond_1, \ldots, \diamond_{i-1}, \theta^*(e)(\overline{\diamond}), \diamond_{i+1}, \ldots, \diamond_n)$$

Intuitivement, la première condition garantit que la taille des objets stockés dans les attributs demeure bornée polynomialement. La nouvelle variable $T$ peut être vue comme un facteur temporel qui permet de prendre en compte le nombre d'itérations dans une commande loop. Ce nombre est borné polynomialement par la taille des objets stockés dans les attributs par définition de la sémantique d'une boucle loop. La seconde condition sur les commandes de type while est semblable à ceci près que le facteur temporel n'apparait plus. Ce phénomène est directement lié au fait que l'on n'ait aucune information sur la terminaison d'une commande de type while.

**Exemple 45.** *Le programme de l'exemple 37 est fraternel puisqu'il admet une sup-interprétation polynomiale* $\theta$ *et qu'il n'a aucune commande de type loop ou de type while.*

**Exemple 46.** *Le programme de l'exemple 38 est fraternel s'il admet une sup-interprétation* $\theta$ *et un poids* $\omega$ *polynomiaux vérifiant les inégalités ci-dessous pour l'unique commande de type loop* $\mathtt{Cm} = \mathtt{loop}\ Y\ \{X_2 := \mathtt{new}\ \ \mathbf{c}(\mathbf{hd}(X_1), X_2); X_1 := \mathbf{tl}(X_1)\}$ *:*

$$\omega_{\mathtt{Cm}}(T+1, \theta^*(X_1), \theta^*(X_2), \theta^*(Y)) \geq \omega_{\mathtt{Cm}}(T, \theta^*(X_1), \theta^*(\mathtt{new}\ \ \mathbf{c}(\mathbf{hd}(X_1), X_2), \theta^*(Y)))$$
$$\omega_{\mathtt{Cm}}(T+1, \theta^*(X_1), \theta^*(X_2), \theta^*(Y)) \geq \omega_{\mathtt{Cm}}(T, \theta^*(\mathbf{tl}(X_1)), \theta^*(X_2), \theta^*(Y))$$

*En prenant* $\theta(\mathbf{c})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2 + 1$, $\theta(\mathbf{hd})(\diamond) = \theta(\mathbf{tl})(\diamond) = \diamond$*, on obtient :*

$$\omega_{\mathtt{Cm}}(T+1, \theta^*(X_1), \theta^*(X_2), \theta^*(Y)) \geq \omega_{\mathtt{Cm}}(T, \theta^*(X_1), 1 + \theta^*(X_1) + \theta^*(X_2), \theta^*(Y)))$$
$$\omega_{\mathtt{Cm}}(T+1, \theta^*(X_1), \theta^*(X_2), \theta^*(Y)) \geq \omega_{\mathtt{Cm}}(T, \theta^*(X_1), \theta^*(X_2), \theta^*(Y))$$

*Finalement, en prenant* $\omega_{\texttt{Cm}}(T, X, Y, Z) = T \times (X+1) + Y + Z$*, on vérifie les inégalités ci-dessus et le programme est donc fraternel.*

**Exemple 47.** *Considérons le programme suivant sur les entiers unaires :*

$$
\begin{array}{l}
\texttt{Class main}\,\{\\
\qquad\qquad \texttt{var}\ X_1;\\
\qquad\qquad \texttt{var}\ X_2;\\
\qquad\qquad \texttt{var}\ X_3;\\
\qquad\qquad \texttt{loop}\ \ X_1\,\{\ X_3 := \texttt{add}(X_3, X_2);\ \}\,;\\
\}
\end{array}
$$

*où* `add` *est un symbole d'opérateur calculant l'addition unaire.*

$\texttt{Cm} = \texttt{loop}\ \ X_1\,\{X_3 := \texttt{add}(X_3, X_2)\}$ *est la seule commande type loop. En appliquant le critère fraternel, il nous faut trouver un poids polynomial* $\omega_{\texttt{Cm}}$ *et une sup-interprétation polynomiale* $\theta$ *tels que :*

$$\omega_{\texttt{Cm}}(T+1, \theta(X_1), \theta(X_2), \theta(X_3)) \geq \omega_{\texttt{Cm}}(T, \theta(X_1), \theta(X_2), \theta^*(\texttt{add}(X_3, X_2)))$$

*avec* $T$ *une nouvelle variable.*

*Puisque* $\theta(\texttt{add})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2$ *est une sup-interprétation du symbole d'opérateur* `add`*, l'inégalité précédente est satisfaite en prenant* $\omega_{\texttt{Cm}}(T, \diamond_1, \diamond_2, \diamond_3) = T \times \diamond_3 + \diamond_1 + \diamond_2$ *et on démontre ainsi que le programme est fraternel.*

### III.3.2 Propriétés des programmes fraternels

**Lemme III.10.** *Étant donné un programme fraternel* $\mathbf{p}$ *de classe principale de la forme* $\texttt{Class main}\ \ \{A; \texttt{Cm}\}$ *possédant* $n$ *attributs* $X_1, \cdots, X_n$*, pour toute commande externe* $\texttt{Cm}_1 \subseteq \texttt{Cm}$*, il existe un polynôme* $P$ *tel que pour tout magasin* $\sigma$ *si* $\langle \texttt{Cm}, \sigma\rangle \downarrow \langle \sigma' \rangle$ *alors :*

$$P(|X_1\sigma|, \ldots, |X_n\sigma|) \geq \max_{i=1..n}(|X_i\sigma'|)$$

*Démonstration.* On commence par attribuer une fonction $P_{\texttt{Cm}}$ de **Max-Poly** à chaque commande externe et minimum `Cm` :

- Si $\texttt{Cm} = \texttt{skip}$ alors on pose $P_{\texttt{Cm}}(\diamond_1, \cdots, \diamond_n) = \max(\diamond_i)$.
- Si $\texttt{Cm} = X := e$ (ou `new` $e$) alors on choisit $P_{\texttt{Cm}}(\diamond_1, \cdots, \diamond_n) = \max(\theta^*(e)(\alpha \times \bar{\diamond}), \alpha \times \diamond_1, \cdots, \alpha \times \diamond_n)$ avec $\alpha$ la constante du lemme III.4.
- Si $\texttt{Cm} = \texttt{loop}\ \ X\,\{\texttt{Cm}_1\}$ alors $P_{\texttt{Cm}}(\diamond_1, \cdots, \diamond_n) = \omega_{\texttt{Cm}}(R(\diamond_1, \cdots, \diamond_n), \alpha \times \bar{\diamond})$ avec $\alpha$ la constante du lemme III.4 et $R$ un polynôme bornant le nombre d'assignations

effectuées durant l'exécution d'une commande de type loop. Un tel nombre est clairement polynomial en la taille des entrées et peut être calculé par analyse statique des programmes.

– Si $\texttt{Cm} = \texttt{while}\ e\,\{\texttt{Cm}_1\}$ alors on prend $P_{\texttt{Cm}}(\diamond_1, \cdots, \diamond_n) = \omega_{\texttt{Cm}}(\alpha \times \diamond_1, \cdots, \alpha \times \diamond_n)$.

Puis, on étend cette assignation aux commandes externes :

– Si $\texttt{Cm} = \texttt{Cm}_1; \texttt{Cm}_2$ alors $P_{\texttt{Cm}}(\overline{\diamond}) = Q_{\texttt{Cm}_2}(Q_{\texttt{Cm}_1}(\max_{i\in\{1,\ldots,n\}}(\diamond_i)))$, où $Q_{\texttt{Cm}}(\diamond)$ est défini comme étant égal à $P_{\texttt{Cm}}(\diamond, \ldots, \diamond)$.
– Si $\texttt{Cm} = \texttt{if}\ e\ \texttt{then}\ \texttt{Cm}_1\ \texttt{else}\ \texttt{Cm}_2$ alors

$$P_{\texttt{Cm}}(\overline{\diamond}) = \max(Q_{\texttt{Cm}_2}(\max_{i\in\{1,\ldots,n\}}(\diamond_i)), Q_{\texttt{Cm}_1}(\max_{i\in\{1,\ldots,n\}}(\diamond_i)))$$

Il nous reste à vérifier que pour toute commande externe $\texttt{Cm}$ si $\langle \texttt{Cm}, \sigma\rangle \downarrow \langle \sigma'\rangle$ alors on a $P_{\texttt{Cm}}(|X_1\sigma|, \ldots, |X_n\sigma|) \geq \max_{i=1..n}(|X_i\sigma'|)$. On commence par le démontrer pour les commandes externes et minimums. Ce résultat est évident dans le cas où $\texttt{Cm} = \texttt{skip}$. À présent, en utilisant la notation $\overline{X}\sigma = X_1\sigma, \ldots, X_n\sigma$ :

– Si $\texttt{Cm} = X_i := e$ (ou $\texttt{new}\ \ e$) et si $\langle e, \sigma\rangle \downarrow \langle v, \sigma'\rangle$, le lemme III.6 garantit que :

$$Q(\theta^*(\overline{X}\sigma)) = \max(\theta^*(\overline{X}\sigma), \theta^*(e)(\theta^*(\overline{X}\sigma)))$$

est une borne supérieure sur $\max(\theta^*(v), \theta^*(\overline{X}\sigma'))$ puisqu'il n'y a pas de paramètre dans l'expression $e$. En appliquant le lemme III.4, on obtient que :

$$\begin{aligned} P_{\texttt{Cm}}(|\overline{X\sigma}|) &= Q(\alpha \times |\overline{X\sigma}|) \\ &\geq Q(\theta^*(\overline{X\sigma})) \\ &\geq \max(\theta^*(v), \theta^*(\overline{X}\sigma')) \\ &\geq \max(|X_1\sigma'|, \ldots, |X_n\sigma'|) \end{aligned}$$

– Si $\texttt{Cm} = \texttt{loop}\ \ X\,\{\texttt{Cm}_1\}$, en combinant le lemme III.6 et le critère fraternel pour les commandes de type loop, on obtient que pour toute commande $X := e$ dans le champ d'une itération loop, si $\langle X := e, \sigma\rangle \downarrow \langle \sigma'\rangle$ alors $\omega_{\texttt{Cm}}(T+1, \theta^*(\overline{X\sigma})) \geq \omega_{\texttt{Cm}}(T, \theta^*(\overline{X\sigma'}))$. Par conséquent, en supposant que le nombre des assignations apparaissant pendant l'exécution d'une telle commande est borné par $R(|X_1\sigma|, \ldots, |X_n\sigma|)$, on en déduit que $\omega_{\texttt{Cm}}(R(|X_1\sigma|, \ldots, |X_n\sigma|), \theta^*(\overline{X\sigma}))$ est une borne supérieure sur $\omega_{\texttt{Cm}}(0, \theta^*(\overline{X\sigma'})$ lorsque $\langle \texttt{loop}\ \ X\,\{\texttt{Cm}_1\}, \sigma\rangle \downarrow \langle \sigma'\rangle$. La combinaison de la propriété sous-terme et du

lemme III.4 nous donne :

$$\begin{aligned}\omega_{\texttt{Cm}}(R(|X_1\sigma|,\dots,|X_n\sigma|),\alpha\times|\overline{X\sigma}|) &\geq \omega_{\texttt{Cm}}(R(|\overline{X\sigma}|),\theta^*(\overline{X\sigma}))\\ &\geq \omega_{\texttt{Cm}}(0,\theta^*(\overline{X\sigma'}))\\ &\geq \omega_{\texttt{Cm}}(0,|\overline{X\sigma'}|)\\ &\geq \max_{i\in\{1,\dots,n\}}(|X_i\sigma'|)\end{aligned}$$

– Si $\texttt{Cm} = \texttt{while}\ e\,\{\texttt{Cm}_1\}$ alors on applique le raisonnement précédent, sans le facteur temporel $T$.

Puis on continue à démontrer ce résultat sur les commandes externes. Supposons, par Hypothèse d'Induction, que nous ayons déjà démontré que la commande `Cm` possède un polynôme $P_{\texttt{Cm}}(\diamond_1,\cdots,\diamond_n)$ vérifiant si $\langle\texttt{Cm},\sigma\rangle\downarrow\langle\sigma'\rangle$ alors $P_{\texttt{Cm}}(|X_1\sigma|,\dots,|X_n\sigma|)\geq \max(|X_1\sigma'|,\dots,|X_n\sigma'|)$. On a :

$$\begin{aligned}Q_{\texttt{Cm}}(\max(|X_1\sigma|,\dots,|X_n\sigma|)) &\geq P_{\texttt{Cm}}(|X_1\sigma|,\dots,|X_n\sigma|)\\ &\geq \max(|X_1\sigma'|,\dots,|X_n\sigma'|)\end{aligned}$$

– Si $\texttt{Cm} = \texttt{Cm}_1;\texttt{Cm}_2$ et $\langle\texttt{Cm},\sigma\rangle\downarrow\langle\sigma'\rangle$ alors il existe un magasin $\sigma''$ tel que $\langle\texttt{Cm}_1,\sigma\rangle\downarrow\langle\sigma''\rangle$ et $\langle\texttt{Cm}_2,\sigma''\rangle\downarrow\langle\sigma'\rangle$. Par Hypothèse d'Induction, on a :

$$Q_{\texttt{Cm}_1}(\max(|X_1\sigma|,\dots,|X_n\sigma|)) \geq \max(|X_1\sigma''|,\dots,|X_n\sigma''|)$$
$$Q_{\texttt{Cm}_2}(\max(|X_1\sigma''|,\dots,|X_n\sigma''|)) \geq \max(|X_1\sigma'|,\dots,|X_n\sigma'|)$$

et, par monotonie des polynômes $Q_{\texttt{Cm}}$ (Ces polynômes sont monotones puisqu'ils sont définis à l'aide de sup-interprétations monotones en utilisant uniquement des fonctions monotones), on obtient :

$$P_{\texttt{Cm}}(\max(|X_1\sigma|,\dots,|X_n\sigma|)) \geq \max(|X_1\sigma'|,\dots,|X_n\sigma'|)$$

– Le dernier cas de figure où $\texttt{Cm} = \texttt{if}\ e\ \texttt{then}\ \texttt{Cm}_1\ \texttt{else}\ \texttt{Cm}_2$ se traite de manière identique.

On a donc construit une fonction $P_{\texttt{Cm}}$ de **Max-Poly** pour toute commande externe `Cm` et il est aisé de trouver un polynôme $P$ vérifiant $P(\diamond_1,\cdots,\diamond_n)\geq P_{\texttt{Cm}}(\diamond_1,\cdots,\diamond_n)$. □

**Théorème III.11.** *Étant donné un programme fraternel* $\mathbf{p}$ *de classe principale de la forme* $\texttt{Class main}\,\{A;\texttt{Cm}\}$ *possédant* $n$ *attributs* $X_1,\cdots,X_n$*, il existe un polynôme* $P$ *tel que pour tout magasin* $\sigma$ *si* $\langle\texttt{Cm},\sigma\rangle\downarrow\langle\sigma'\rangle$ *alors*

$$P(|X_1\sigma|,\dots,|X_n\sigma|)\geq \max_{i=1..n}(|X_i\sigma'|)$$

*Démonstration.* D'après le lemme III.10, toute commande externe de la classe principale d'un programme fraternel a un calcul dont la taille est bornée par un polynôme. Il suffit de constater que la commande `Cm` d'une classe principale est une commande externe. □

## III.4 Synthèse de sup-interprétations

Le critère fraternel permet d'étudier la complexité d'un très grand nombre de programmes. Cependant, pour pouvoir être appliqué, il requiert la connaissance d'une borne supérieure, la sup-interprétation, sur les calculs effectués par certaines méthodes. Par conséquent, il est important de pouvoir développer un critère sur les méthodes d'une classe afin de construire une sup-interprétation de ces méthodes.

### III.4.1 Poids d'une méthode

**Définition III.12.** *Une méthode de la forme* $\texttt{f}(\overline{x})\,\{\texttt{Cm ; return } X;\}$ *est dite :*

- *laborieuse s'il existe une commande de type while* $\texttt{Cm}_1$ *telle que* $\texttt{Cm}_1 \sqsubseteq \texttt{Cm}$
- *élégante dans le cas contraire.*

**Définition III.13** (Poids d'une méthode)**.** *Étant donnée une classe* C *possédant* $n$ *attributs, le poids* $\omega_{\texttt{D}}$ *d'une méthode* D *de la classe* C *est une fonction totale* $\omega_{\texttt{D}}$ *de* $(\mathbb{R}^+)^{n+m+1}$ *dans* $\mathbb{R}^+$*, si* D *est une méthode élégante, et une fonction totale* $\omega_{\texttt{D}}$ *de* $(\mathbb{R}^+)^{n+m}$ *dans* $\mathbb{R}^+$*, si* D *est une méthode laborieuse, qui satisfait* $\omega_{\texttt{D}}$ *est monotone et possède la propriété sous-terme (Cf. Définition III.8). Un poids* $\omega_{\texttt{D}}$ *est dit polynomial si* $\omega_{\texttt{D}}$ *est une fonction de* **Max-Poly** $\{\mathbb{R}^+\}$.

### III.4.2 Méthodes fraternelles

**Définition III.14.** *Étant donnée une classe* C *de* $n$ *attributs* $\overline{X} = X_1, \cdots, X_n$*, une méthode* D *de* C *et la forme* $\texttt{f}(\overline{x})\,\{\texttt{Cm ; return } X_i;\}$ *est* ***fraternelle*** *s'il existe une sup-interprétation polynomiale et additive* $\theta$ *et un poids de méthode polynomial* $\omega_{\texttt{D}}$ *tels que* $\theta(X_j) = \diamond_j$, $\theta(x_k) = \square_k$ *et :*

1. *Si* D *est élégante, on a :*

   *Pour toute expression* $e = X_j.\texttt{f}(e_1, \cdots, e_m)$ *apparaissant dans* Cm *:*

$$\omega_{\texttt{D}}(T+1, \overline{\square}, \overline{\diamond}) \geq \omega_{\texttt{D}}(T, \overline{\square}, \diamond_1, \ldots, \diamond_{j-1}, \theta^*(e)(\overline{\square}, \overline{\diamond}), \diamond_{j+1}, \ldots, \diamond_n)$$

   *Pour toute commande* $X_i := e \sqsubseteq \texttt{Cm}$*, on a :*

$$\omega_{\texttt{D}}(T+1, \overline{\square}, \overline{\diamond}) \geq \omega_{\texttt{D}}(T, \overline{\square}, \diamond_1, \ldots, \diamond_{i-1}, \theta^*(e)(\overline{\square}, \overline{\diamond}), \diamond_{i+1}, \ldots, \diamond_n)$$

   *avec* $T$ *une nouvelle variable.*

2. *Si* D *est laborieuse, on a :*

   *Pour toute expression* $e = X_j.\texttt{f}(e_1, \cdots, e_m)$ *apparaissant dans* Cm *:*

$$\omega_{\texttt{D}}(\overline{\square}, \overline{\diamond}) \geq \omega_{\texttt{D}}(\overline{\square}, \diamond_1, \ldots, \diamond_{j-1}, \theta^*(e)(\overline{\square}, \overline{\diamond}), \diamond_{j+1}, \ldots, \diamond_n)$$

*Pour toute commande $X_i := e \sqsubseteq$ `Cm`, on a :*

$$\omega_{D}(\overline{\Box}, \overline{\diamond}) \geq \omega_{D}(\overline{\Box}, \diamond_1, \ldots, \diamond_{i-1}, \theta^*(e)(\overline{\Box}, \overline{\diamond}), \diamond_{i+1}, \ldots, \diamond_n)$$

**Théorème III.15.** *Étant données une classe $C$ de $n$ attributs $\overline{X}$ et une méthode fraternelle $D = \mathtt{f}(\overline{x})\,\{\mathtt{Cm}\ ;\ \mathtt{return}\ X_i;\}$ de $C$ de sup-interprétation $\theta$ telle que $\theta(X_j) = \diamond_j$ et $\theta(x_k) = \Box_k$, l'une des deux propriétés suivantes est vérifiée :*
- *Soit $D$ est laborieuse et $\omega_{D}(\overline{\Box}, \overline{\diamond})$ est une sup-interprétation de `f`.*
- *Soit $D$ est élégante et si $R(\overline{\diamond})$ est une borne supérieure sur le nombre d'assignations apparaissant pendant l'exécution de `Cm` alors $\omega_{D}(R(\overline{\diamond}), \overline{\Box}, \overline{\diamond})$ est une sup-interprétation de `f`.*

*Démonstration.* La preuve est identique à celle du théorème III.11. La seule différence étant que les paramètres peuvent apparaitre dans les expressions. Le résultat en découle puisque la propriété sous-terme de $\omega_{D}$, $\forall j \in \{1, \ldots, n\}$, permet l'obtention d'une borne supérieure sur la taille de l'objet stocké en $X_j$. □

La restriction imposant qu'une commande `loop` $X\,\{Cm\}$ ne puisse pas écrire dans l'attribut $X$ implique que le polynôme $R(\overline{\diamond})$ peut être calculé par analyse statique d'un programme et que, par conséquent, le critère sur les méthodes fraternelles donne des heuristiques permettant de trouver la sup-interprétation d'une classe. En effet, si nous arrivons à trouver un poids polynomial pour toute méthode d'une classe alors nous obtenons une sup-interprétation de la méthode.

**Exemple 48.** *Considérons la méthode de l'exemple 37 :*

$$D = \mathtt{move}(x, y)\,\{X := \mathtt{add}(X, x); Y := \mathtt{add}(Y, y)\ ;\ \mathtt{return}\ X;\}$$

*Si l'on considère que la méthode `add` correspond à l'addition unaire, on peut choisir sa sup-interprétation comme étant $\theta(\mathtt{add})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2$. Afin de vérifier que cette méthode est fraternelle, il nous faut trouver un poids polynomial $\omega_{D}$ satisfaisant :*

$$\omega_{D}(T+1, \Box_1, \Box_2, \diamond_1, \diamond_2) \geq \omega_{D}(T, \Box_1, \Box_2, \theta^*(\mathtt{add}(\diamond_1, \Box_1)), \diamond_2)$$
$$\omega_{D}(T+1, \Box_1, \Box_2, \diamond_1, \diamond_2) \geq \omega_{D}(T, \Box_1, \Box_2, \diamond_1, \theta^*(\mathtt{add}(\diamond_2, \Box_2)))$$

*avec $\theta(X) = \Box_1$, $\theta(Y) = \Box_2$, $\theta(x) = \diamond_1$ et $\theta(y) = \diamond_2$. Puisque $\theta^*(\mathtt{add}(X, x)) = \theta(\mathtt{add})(\theta(x), \theta(X)) = \theta(x) + \theta(X) = \Box_1 + \diamond_1$, il nous faut vérifier les inégalités suivantes :*

$$\omega_{D}(T+1, \Box_1, \Box_2, \diamond_1, \diamond_2) \geq \omega_{D}(T, \Box_1, \Box_2, \Box_1 + \diamond_1, \diamond_2)$$
$$\omega_{D}(T+1, \Box_1, \Box_2, \diamond_1, \diamond_2) \geq \omega_{D}(T, \Box_1, \Box_2, \diamond_1, \Box_2 + \diamond_2)$$

*En prenant* $\omega_D(T, \square_1, \square_2, \diamond_1, \diamond_2) = \max(T \times \square_1 + \diamond_1, T \times \square_2 + \diamond_2)$, *on démontre que cette méthode est fraternelle. Nous savons qu'au plus deux assignations sont exécutées dans la commande de la méthode. Par conséquent,* $R(\diamond_1, \cdots, \diamond_n) = 2$ *et, d'après le théorème III.15, on obtient que* $\max(2 \times \square_1 + \diamond_1, 2 \times \square_2 + \diamond_2)$ *est une sup-interprétation de* `move`.

## III.5 Caractérisations de classes de complexité

### III.5.1 Une caractérisation de FPspace

Dans cette section, nous donnons une caractérisation de l'ensemble des fonctions calculables en espace polynomial à l'aide du critère fraternel. Le critère fraternel nous garantit que la taille d'un magasin demeure bornée polynomialement par la taille des entrées durant un calcul mais ne donne aucune information sur la terminaison des programmes orientés-objet. Par conséquent, nous allons restreindre notre étude à des programmes terminants et utilisant des symboles d'opérateur **op** dont la fonction calculée correspondante $[\![\mathbf{op}]\!]$ est calculable par une Machine de Turing en espace polynomial.

Nous restreignons les calculs autorisés pour les symboles d'opérateur en nous inspirant de la terminologie de [NW06] :

**Définition III.16.** *Un symbole d'opérateur* **op** *de Op est admissible en espace si* $[\![\mathbf{op}]\!]$ *peut être simulé par une Machine de Turing en espace polynomial.*

**Théorème III.17.** *L'ensemble des fonctions calculées par des programmes fraternels qui terminent et possédant des symboles d'opérateur admissibles en espace est exactement l'ensemble des fonctions calculables en espace polynomial.*

*Démonstration.* Nous commençons par montrer que l'ensemble des fonctions calculées par des programmes fraternels qui terminent et possédant des symboles d'opérateur admissibles en espace est inclus dans FPSPACE. Supposons que $\langle \mathtt{Cm}, \sigma \rangle \downarrow \langle \sigma' \rangle$. Une telle évaluation existe puisque le programme termine. D'après le théorème III.11, il existe un polynôme $Q$ tel que pour toute commande $\mathtt{Cm}_1 \sqsubseteq \mathtt{Cm}$, si $\langle \mathtt{Cm}_1, \sigma \rangle \downarrow \langle \rho \rangle$ alors $Q(|X_1\sigma|, \ldots, |X_n\sigma|) \geq \max_{i=1..n}(|X_i\rho|)$. De sorte que le programme calcule un nombre fini de magasins distincts dont la taille est bornée polynomialement par la taille des entrées, où la taille d'un magasin est la somme des tailles de tous les objets qui y sont stockés. Puisqu'un magasin peut être vu comme une configuration d'une Machine de Turing, le calcul d'un tel programme peut être simulé par une Machine de Turing en espace polynomial.

Réciproquement, supposons que nous disposions d'une Machine de Turing à un ruban et $n$ états sur un alphabet $\Sigma$ calculant en espace borné par un polynôme $P$. On commence par ajouter à notre programme un attribut $X$ dans lequel on calcule le polynôme $P$ à l'aide de commandes loop imbriquées sur un alphabet unaire. Nous simulons chaque symbole de l'alphabet par un identifiant de classe distinct. Le ruban est codé par deux attributs $Y_1$ et $Y_2$ qui représentent la partie située à gauche de la tête de lecture et la partie située à droite de la tête de lecture. Nous supposerons que le premier symbole sur la partie droite du ruban correspond à la position courante de la tête de lecture. Puisque la taille de l'objet stocké dans $X$ est une borne supérieure sur l'espace utilisé pendant un calcul de la Machine de Turing, pour chacun des deux attributs du ruban, nous ajoutons la commande suivante $Y_i := X$. Cette commande permet d'initialiser chacun des objets en lui attribuant l'espace maximal autorisé. Chaque état est codé par une classe distincte. La classe des rubans possède une méthode $t.\mathtt{transition}(s)$ qui, étant donné un état $s$ en entrée, retourne un état, écrit un nouveau symbole sur le ruban $t$ et change la position courante de la tête de lecture relativement au mouvement à son mouvement dans la Machine de Turing. Cette méthode permet de simuler les transitions d'une Machine de Turing. Nous choisissons la sup-interprétation de tout identifiant de classe $\mathtt{C} \in \mathtt{Class}$ comme étant additive et de la forme $\theta(\mathtt{C})(\diamond_1, \cdots, \diamond_n) = \sum_{j=1}^{n} \diamond_j + 1$. La méthode `transition` n'augmente pas la taille du ruban puisqu'elle se contente de réécrire les symboles. Par conséquent, on peut choisir sa sup-interprétation comme étant égale à $\theta(\mathtt{transition})(T, S) = T$. Pour finir, on représente la Machine de Turing par une classe principale de trois attributs $T_1, S$ et $X$. L'attribut $T_1$ contient l'objet qui code le ruban, $S$ correspond à l'objet codant l'état courant et $X$ stocke l'objet dont la taille correspond à la simulation du polynôme $P$ décrite ci-dessus. Le programme simulant la machine s'écrit sous la forme suivante :

$$\mathtt{Cm} = \mathtt{while}\ S \neq \mathtt{new}\ q_f()\, \{\ldots; S := T_1.\mathtt{transition}(S); \ldots\}$$

où l'expression $S \neq \mathtt{new}\ q_f()$ vérifie que l'état courant est différent de l'identifiant de classe $q_f$ qui code l'état final de la machine. Cette commande est de type while de sorte qu'il nous faut trouver un poids polynomial satisfaisant :

$$\begin{aligned} \omega_{\mathtt{Cm}}(\theta(T_1), \theta(X), \theta(S)) &\geq \omega_{\mathtt{Cm}}(\theta^*(T_1.\mathtt{transition}(S)), \theta(X), \theta(S)) \\ &= \omega_{\mathtt{Cm}}(\theta(T_1), \theta(X), \theta(S)) \\ &\text{et} \\ \omega_{\mathtt{Cm}}(\theta(T_1), \theta(X), \theta(S)) &\geq \omega_{\mathtt{Cm}}(\theta(T_1), \theta(X), \theta^*(T_1.\mathtt{transition}(S))) \\ &= \omega_{\mathtt{Cm}}(\theta(T_1), \theta(X), \theta(T_1)) \end{aligned}$$

En prenant $\omega_{\texttt{Cm}}(\diamond_1, \diamond_2, \diamond_3) = \max(\diamond_1, \diamond_2, \diamond_3)$, on obtient que le programme est fraternel. □

### III.5.2 Une caractérisation de FPtime

Dans cette section, nous proposons une caractérisation de l'ensemble des fonctions calculables en temps polynomial en utilisant le critère fraternel. Puisque le critère fraternel ne donne aucune indication sur la terminaison des programmes orientés-objet, il nous faut restreindre de manière drastique la puissance de calcul des programmes considérés :

**Définition III.18.** *Un programme* $\mathbf{p}$ *est strictement fraternel s'il est fraternel et qu'il ne possède aucune commande de type while et aucune méthode laborieuse.*

**Lemme III.19.** *Un programme strictement fraternel termine et son exécution correspond à un nombre d'assignations borné polynomialement en la taille des entrées.*

*Démonstration.* Une commande externe qui n'est pas de type while termine et son exécution correspond à l'exécution d'un nombre d'assignations borné polynomialement en la taille des entrées, puisqu'une commande de type loop est exécutée un nombre polynomial de fois.

Nous posons des restrictions sur les calculs autorisés pour les symboles d'opérateur :

**Définition III.20.** *Un symbole d'opérateur* **op** *de Op est admissible en temps si* $[\![\mathbf{op}]\!]$ *peut être simulé par une Machine de Turing en temps polynomial.*

**Théorème III.21.** *L'ensemble des fonctions calculées par des programmes strictement fraternels et admettant des symboles d'opérateur admissibles en temps est exactement l'ensemble des fonctions calculables en temps polynomial.*

*Démonstration.* On commence par montrer que l'ensemble des fonctions calculées par des programmes strictement fraternels est inclus dans FPtime. Supposons que $\langle \texttt{Cm}, \sigma\rangle \downarrow \langle \sigma'\rangle$. Une telle évaluation existe puisque le programme termine d'après le lemme III.19. De plus, ce lemme nous garantit que le nombre $n$ des assignations est borné par $P(|X\sigma|)$, avec $P$ un polynôme donné. D'après le théorème III.11, il existe un polynôme $Q$ tel que pour toute commande $\texttt{Cm}_1 \sqsubseteq \texttt{Cm}$ si $\langle \texttt{Cm}_1, \sigma\rangle \downarrow \langle \rho\rangle$ alors $Q(|X_1\sigma|, \dots, |X_n\sigma|) \geq \max_{i=1..n}(|X_i\rho|)$. De sorte que le programme calcule un nombre polynomial de magasins distincts stockant des objets dont les tailles sont toutes bornées polynomialement par la taille des entrées. Finalement, un magasin peut être vu comme une configuration d'une Machine de Turing et le calcul peut être simulé par une Machine de Turing en temps polynomial.

Réciproquement, supposons que nous disposions d'une Machine de Turing à un ruban et $n$ états calculant sur un alphabet $\Sigma$ et s'exécutant en temps borné par un polynôme $P$. Nous commençons par ajouter à notre programme un attribut $X$ dans lequel on calcule le polynôme $P$ à l'aide de commandes loop imbriquées. Nous simulons chaque symbole de l'alphabet par un identifiant de classe distinct. Le ruban est codé par deux attributs qui représentent les parties du ruban à gauche et à droite de la tête de lecture, la position courante de la tête de lecture correspondant au premier symbole stocké dans l'attribut représentant la partie droite du ruban. De plus, on code chaque état par un identifiant de classe distinct. La classe des rubans possède une méthode $t.\texttt{transition}(s)$ qui, pour un état $s$ donné en entrée, retourne un état, écrit un nouveau symbole sur le ruban $t$ et change la position courante de la tête de lecture relativement au mouvement de la tête de lecture dans la Machine de Turing correspondante. Cette méthode permet de simuler les transitions d'une Machine de Turing. On choisit la sup-interprétation de tout identifiant de classe $\texttt{C} \in \texttt{Class}$ comme étant additive de la forme $\theta(\texttt{C})(\diamond_1, \cdots, \diamond_n) = \sum_{j=1}^{n} \diamond_j + 1$. La méthode $\texttt{transition}$ augmente la taille d'un ruban d'au plus un symbole. Par conséquent, on peut choisir sa sup-interprétation comme étant égale à $\theta(\texttt{transition})(T, S) = T + 1$. À présent, on représente la Machine de Turing par une classe principale possédant trois attributs $T_1, X$ et $S$. L'attribut $T_1$ contient l'objet qui code le ruban, $S$ correspond à l'objet qui représente l'état courant et $X$ stocke un objet dont la taille correspond à la simulation du polynôme $P$ décrite ci-dessus. Le programme simulant la machine a la forme suivante :

$$\texttt{Cm} = \texttt{loop}\ \ X\,\{\ldots; S := T_1.\texttt{transition}(S); \ldots\}$$

Cette commande est de type loop de sorte qu'il nous faut trouver un poids polynomial vérifiant :

$$\begin{aligned}
\omega_{\texttt{Cm}}(T+1, \theta(T_1), \theta(X), \theta(S)) &\geq \omega_{\texttt{Cm}}(T, \theta^*(T_1.\texttt{transition}(S)), \theta(X), \theta(S)) \\
&= \omega_{\texttt{Cm}}(T, \theta(T_1) + 1, \theta(X), \theta(S)) \\
&\text{et} \\
\omega_{\texttt{Cm}}(T+1, \theta(T_1), \theta(X), \theta(S)) &\geq \omega_{\texttt{Cm}}(T, \theta(T_1), \theta(X), \theta^*(T_1.\texttt{transition}(S))) \\
&= \omega_{\texttt{Cm}}(T, \theta(T_1), \theta(X), \theta(T_1) + 1)
\end{aligned}$$

Le programme est donc strictement fraternel en choisissant $\omega_{\texttt{Cm}}(T, T_1, X, S) = T + \max(T_1, X, S)$. □

## III.6 Conclusion

Nous avons ainsi obtenu une extension de nos outils à des langages orientés-objet. On constate que l'outil s'adapte correctement à différents langages de programmation, prenant en compte des phénomènes nouveaux comme les effets de bord pour les langages objet. Une conséquence de cette flexibilité est que, quelque soit le langage en considération, les critères de polynomialité ainsi que les caractérisations des classes de complexité polynomiales sont très similaires. Les travaux contenus dans ce chapitre sont les plus récents et sont très certainement perfectibles. Par exemple, on aurait pu prendre en compte la notion de super-classe. De plus, on pourrait essayer de les adapter à des langages objet plus évolués, par exemple, à des langages incluant des traits [LS07]. Cependant, ils ont le mérite de donner une première approche simple et cohérente de l'étude de la complexité des programmes objets. Il va de soi que les résultats obtenus peuvent être étendus sans grande difficulté à un langage plus évolué. Par exemple, la notion d'héritage peut être assimilée à des classes partageant des sup-interprétations identiques.

# Conclusion

Our progress as a nation can be no swifter than our progress in education. The human mind is our fundamental resource.

*(J.F. Kennedy)*

Dans ce document, nous avons essayé de donner une vue d'ensemble des travaux effectués concernant le contrôle des ressources à l'aide d'interprétations sémantiques à valeur sur des fonctions max-polynomiales. Après avoir introduit les quasi-interprétations, nous avons étudié les différentes propriétés et caractérisations obtenues à l'aide de cette notion. Nous avons étudié le problème de la synthèse de cet outil, puis, nous avons abordé la question de sa modularité. Après avoir mentionné différentes extensions à des langages de programmation réactif, bytecode ou d'ordre supérieur, nous avons introduit la sup-interprétation. Cette notion généralise la quasi-interprétation et a été adaptée dans des critères de contrôle des ressources afin d'étudier la complexité d'un plus grand nombre d'algorithmes comme les algorithmes sur les streams ou les algorithmes de type « diviser pour régner ». Elle a aussi été combinée à différents critères de terminaison comme les ordres RPO, les paires de dépendance ou le size-change principle. En outre, après avoir caractérisé des petites classes de complexité parallèles, nous avons donné quelques heuristiques permettant de synthétiser des sup-interprétations sans la propriété sous-terme, c'est-à-dire des sup-interprétations qui ne sont pas des quasi-interprétations. Enfin, dans un dernier chapitre, nous avons adapté les sup-interprétations à des langages orientés-objet, obtenant ainsi différents critères pour contrôler les ressources d'un programme et de ses méthodes.

Ces travaux mènent à de nombreuses autres pistes de recherche. En particulier, il serait intéressant d'étudier les propositions suivantes :

- Étendre l'étude des langages objet effectuée à un langage plus général incluant l'héritage, des pointeurs ou, encore, des fonctions récursives.
- Étudier les relations entre les interprétations abstraites et les sup-interprétations. En effet, de nombreuses techniques de contrôle de propriétés des langages impé-

ratifs de la famille du C utilisent les interprétations abstraites. En pratique, les interprétations abstraites utilisent des domaines de polyèdres et des intervalles afin de borner les valeurs contenues dans une variable au cours de l'exécution d'un programme. Puisque la connaissance d'une sup-interprétation nous donne un intervalle de confiance dans lequel une valeur oscille au cours de l'exécution d'un programme, il semble donc que les deux notions sont très proches. Une étude comparative serait donc nécessaire.

- Améliorer la synthèse de quasi-interprétations et de sup-interprétations. En pratique, un logiciel appelé CROCUS, disponible à l'adresse `http://libresource.inria.fr//projects/crocus`, permet déjà de synthétiser des quasi-interprétations. Afin de valoriser cette étude, il semble nécessaire d'améliorer les performances de ce logiciel en trouvant des heuristiques plus rapides pour la synthèse de quasi-interprétations, puis, en étendant son application aux sup-interprétations.
- Essayer d'étendre la sup-interprétation à la programmation réactive de manière élégante. En effet, la sup-interprétation a été étendue à des programmes incluant des données infinies comme des streams. De tels programmes s'apparentent fortement à des programmes réactifs, puisque les programmes réactifs maintiennent une interaction permanente avec leur environnement, ce qui nous incite donc à essayer d'adapter notre étude à ce paradigme de programmation.
- Comparer ou combiner notre approche avec les différentes autres approches de la communauté ICC citées dans l'introduction. En particulier, la combinaison d'une variante des sup-interprétations et de la logique linéaire semble envisageable. En effet, la notion de poids dans les preuves [Laf04] semble similaire à la notion d'interprétation polynomiale. Il serait donc intéressant de voir si les restrictions logiques de la logique linéaire peuvent être remplacée par des systèmes de contraintes sur les preuves.

# Bibliographie


[AC96] M. Abadi et L. Cardelli – *A Theory of Objects*, Springer, 1996. 136

[ACGDZJ04] R. Amadio, S. Coupet-Grimal, S. Dal-Zilio et L. Jakubiec – « A functional scenario for bytecode verification of resource bounds. », *CSL 2004*, Lecture Notes in Computer Science, vol. 3210, Springer, 2004, p. 265–279. 13, 67, 75

[Ack28] W. Ackermann – « Zum Hilbertschen Aufbau der reellen Zahlen », *Math. annalen* **99** (1928), p. 118–133. 83

[ADZ04] R. Amadio et S. Dal-Zilio – « Resource control for synchronous cooperative threads. », *CONCUR 2004*, Lecture Notes in Computer Science, vol. 3170, Springer, 2004, p. 68–82. 13, 67, 68, 75, 90

[AG00] T. Arts et J. Giesl – « Termination of term rewriting using dependency pairs », *Theoretical Computer Science* **236** (2000), p. 133–178. 27, 74, 77, 103, 104

[AK03] H. Anderson et S.C. Khoo – « Affined-based size-change termination », *APLAS 2003*, Lecture Notes in Computer Science, vol. 2895, Springer, 2003, p. 122–140. 107

[Ama05] R. Amadio – « Synthesis of max-plus quasi-interpretations », *Fundamenta Informaticae* **65** (2005), no. 1-2, p. 29–60. 13, 34, 35, 41, 42, 101

[Ave06] J. Avery – « Size-change termination and bound analysis », *FLOPS 2006*, Lecture Notes in Computer Science, vol. 3945, Springer, 2006, p. 192–207. 107

[BC80] A. Borodin et S. Cook – *A time-space tradeoff for sorting on a general sequential model of computation*, ACM Press New York, NY, USA, 1980. 112

[BC92] S. Bellantoni et S. Cook – « A new recursion-theoretic characterization of the poly-time functions », *Computational Complexity* **2** (1992), p. 97–110. 8, 11

[BCMT99] G. Bonfante, A. Cichon, J.-Y. Marion et H. Touzet – « Complexity classes and rewrite systems with polynomial interpretation », *CSL'98*, Lecture Notes in Computer Science, vol. 1584, 1999, p. 372–384. 58

[Ben01] R. Benzinger – « Automated complexity analysis of Nuprl extracted programs », *Journal of Functional Programming* **11** (2001), no. 1, p. 3–31. 12

[BIS90] D. Barrington, N. Immerman et H. Straubing – « On uniformity within NC. », *Journal of Computer System Science* **41** (1990), no. 3, p. 274–306. 112

[BKMO06] G. Bonfante, R. Kahle, J.-Y. Marion et I. Oitavem – « Towards an implicit characterization of $NC^k$ », *CSL'06*, Lecture Notes in Computer Science, 2006. 114

[Blo94] S. Bloch – « Function-algebraic characterizations of log and polylog parallel time », *Computational complexity* **4** (1994), no. 2, p. 175–205. 113, 127

[Blu67] M. Blum – « A machine-independent theory of the complexity of recursive functions », *Journal of the ACM* **14** (1967), p. 322–336. 5, 80

[BM04] P. Baillot et V. Mogbil – « Soft lambda-calculus : a language for polynomial time computation », *FOSSACS 2004*, Lecture Notes in Computer Science, vol. 2987, Springer, 2004, p. 27–41. 8

[BMM01] G. Bonfante, J.-Y. Marion et J.-Y. Moyen – « On lexicographic termination ordering with space bound certifications », *PSI 2001*, Lecture Notes in Computer Science, vol. 2244, Springer, Jul 2001. 11, 24, 31, 33, 74

[BMM07] G. Bonfante, J.Y. Marion et J.Y. Moyen – « Quasi-interpretations, a way to control resources », *Theoretical Computer Science* (2007). 11, 12, 13, 24, 28, 31, 74, 107

[BMMP05] G. Bonfante, J.-Y. Marion, J.-Y. Moyen et R. Péchoux – « Synthesis of quasi-interpretations », *Workshop on Logic and Complexity in Computer Science, LCC 2005*, 2005, http://hal.inria.fr. 9, 13, 34, 38

[BMP06] G. Bonfante, J.-Y. Marion et R. Péchoux – « A characterization of alternating log time by first order functional programs », *LPAR 2006*, Lecture Notes in Artificial Intelligence, vol. 4246, Springer, 2006, p. 90–104. 10, 76, 113, 120, 121

[BMP07] G. Bonfante, J.Y. Marion et R. Péchoux – « Quasi-interpretation synthesis by decomposition », *ICTAC 2007*, Lecture Notes in Computer Science, vol. 4711, Springer, 2007. 9, 13, 49, 58, 70

[BN98] F. Baader et T. Nipkow – *Term rewriting and all that*, Cambridge University Press, 1998. 14

[Bon00] G. Bonfante – « Constructions d'ordres, analyse de la complexité », Thèse, Institut National Polytechnique de Lorraine, 2000. 8

[Bor77] A. Borodin – « On Relating Time and Space to Size and Depth », *SIAM Journal on Computing* **6** (1977), p. 733. 112

[Bus87] S. Buss – « The Boolean formula value problem is in ALOGTIME », *STOC'87* (1987), p. 123–131. 112

[BW88] R. Bird et P. Wadler – *Introduction to functional programming*, Prentice-Hall, New York, NY, 1988. 87

[Car56] R. Carnap – *Meaning and necessity*, University of Chicago Press Chicago, 1956. 12

[CC77] P. Cousot et R. Cousot – « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints », *POPL'77* (1977), p. 238–252. 141

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, X. Rival et autres – « The ASTREE analyzer », *ESOP 2005*, Lecture Notes in Computer Science, vol. 3444, Springer, 2005, p. 21–30. 141

[CKS81] A. Chandra, D. Kozen et L. Stockmeyer – « Alternation », *Journal of the ACM* **28** (1981), p. 114–133. 107, 110

[CL90] K.J. Compton et C. Laflamme – « An algebra and a logic for NC », *Inf. Comput.* **87** (1990), no. 1/2, p. 240–262. 113

[Clo89] P. Clote – « Sequential, machine-independent characterizations of the parallel complexity classes ALOGTIME, $AC^k$, $NC^k$ and `NC` », *Workshop on Feasible Math.* (R. Buss et P. Scott, éds.), Birkhäuser, 1989, p. 49–69. 121, 123, 125

[Clo95] — , « Computational models and function algebras », *LCC'94*, Lecture Notes in Computer Science, vol. 960, 1995, p. 98–130. 113

[CLR90] T. Cormen, C. Leiserson et R. Rivest – *Introduction to Algorithms*, MIT Press, 1990. 12

[CMMU03] E. Contejean, C. Marche, B. Monate et X. Urbain – « Proving Termination of Rewriting with CiME », *WST* (2003), p. 71–73. 105

[Cob62] A. Cobham – « The intrinsic computational difficulty of functions », *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science* (Amsterdam) (Y. Bar-Hillel, éd.), North-Holland, 1962, p. 24–30. 7, 123

[Col75] G. E. Collins – « Quantifier elimination for real closed fields by cylindrical algebraic decomposition », *2nd GI Conference on Automata Theory and Formal Languages*, Lecture Notes in Computer Science, vol. 33, 1975. 40

[Coo85] S.A. Cook – « A taxonomy of problems with fast parallel algorithms », *Information and Control* **64** (1985), no. 1-3, p. 2–21. 111, 113

[Dab07] F. Dabrowski – « Programmation réactive synchrone. langage et contrôle des ressources », Thèse, Université de Paris VII, 2007. 69

[Dan06] O. Danvy – « An analytical approach to programs as data objects », 2006, Doctor Scientarum degree in Computer Science. BRICS. Departement of Computer Science. University of Aarhus. 69

[DE99] S. Drossopoulou et S. Eisenbach – « Describing the semantics of Java and proving type soundness », *Formal Syntax and Semantics of Java* (1999), p. 41–82. 136, 140

[Der82] N. Dershowitz – « Orderings for term-rewriting systems », *Theoretical Computer Science* **17** (1982), no. 3, p. 279–301. 12, 26

[Der94] — , « Hierarchical termination », *CTRS'94*, Lecture Notes in Computer Science, vol. 968, Springer, 1994, p. 89–105. 60

[DJ90] N. Dershowitz et J-P Jouannaud – « Handbook of theoretical computer science vol.B », ch. Rewrite systems, p. 243–320, Elsevier Science Publishers B. V. (NorthHolland), 1990. 14

[DZG05] S. Dal-Zilio et R. Gascon – « Resource bound certification for a tail-recursive virtual machine », *APLAS 2005*, Lecture Notes in Computer Science, vol. 3780, Springer, 2005, p. 247–263. 13, 68

[Fag73] R. Fagin – « Generalized first-order spectra and polynomial-time recognizable sets », *Complexity of Comput., Proc. Symp. appl. Math., New York City* (1973). 8

[FM03] S.G. Frankau et A. Mycroft – « Stream processing hardware from functional language specifications », *HICSS 36*, IEEE, 2003. 95

[Fre92] G. Frege – « Über Sinn und Bedeutung [On Sense and Reference] », *Zeitschrift für Philosophie und philosophische Kritik* **100** (1892), p. 25–50. 11

[Gir87] J.-Y. Girard – « Linear logic », *Theoretical Computer Science* **50** (1987), p. 1–102. 8

[Gir98] — , « Light linear logic », *Information and Computation* **143** (1998), no. 2, p. 175–204. 8

[Goe92] A. Goerdt – « Characterizing complexity classes by general recursive definitions in higher types », *Information and Computation* **101** (1992), no. 2, p. 202–218. 8

[Gra92] B. Gramlich – « Generalized sufficient conditions for modular termination of rewriting », *ICALP'92*, vol. 632, Springer, 1992, p. 53–68. 49

[GRDR07] M. Gaboardi et S. Ronchi Della Rocca – « A soft type assignment system for $\lambda$-calculus », *CSL 2007* **4646** (2007), p. 253–267. 8

[GSKT06] J. Giesl, P. Schneider-Kamp et R. Thiemann – « AProVE 1.2 : Automatic termination proofs in the dependency pair framework », *IJCAR 2006* **4130** (2006), p. 281–286. 105

[GSS92] J.Y. Girard, A. Scedrov et P. Scott – « Bounded linear logic », *Theoretical Computer Science* **97** (1992), no. 1, p. 1–66. 8

[Gur83] Y. Gurevich – « Algebras of feasible functions », *FOCS 83*, IEEE Computer Society Press, 1983, p. 210–214. 8

[HM04] N. Hirokawa et A. Middeldorp – « Tyrolean termination tool », *Technical Report AIB-2004-07, RWTH* (2004), p. 59–62. 105

[Hof92] D. Hofbauer – « Termination proofs with multiset path orderings imply primitive recursive derivation lengths », *Theoretical Computer Science* **105** (1992), no. 1, p. 129–140. 27

[Hof00] M. Hofmann – « Programming languages capturing complexity classes », *ACM SIGACT News* **31** (2000), no. 1, p. 31–42. 8

[Hof02] M. Hofmann – « The strength of Non-Size Increasing computation », *POPL'02*, 2002, p. 260–269. 12

[HRS90] J. Heintz, M.-F. Roy et P. Solerno – « Sur la complexité du principe de Tarski-Seidenberg », *Bulletin de la S.M.F., tome 118* (1990), p. 101–126. 40

[Hue80] G. HUET – « Confluent reductions : Abstract properties and applications to term rewriting systems », *Journal of the ACM* **27** (1980), no. 4, p. 797–821. 14

[Imm99] N. IMMERMAN – *Descriptive Complexity*, Springer, 1999. 8

[IPW01] A. IGARASHI, B.C. PIERCE et P. WADLER – « Featherweight Java : A Minimal Core Calculus for Java and GJ », *ACM Transactions on Programming Languages and Systems* **23** (2001), no. 3, p. 396–450. 136, 140

[Jon97] N. D. JONES – *Computability and complexity, from a programming perspective*, MIT press, 1997. 6, 28

[Jon00] — , « The expressive power of higher order types or, life without cons », *Journal of Functional Programming* **11** (2000), no. 1, p. 55–94. 8, 12

[KJ06] L. KRISTIANSEN et N.D. JONES – « The flow of data and the complexity of algorithms », *New Computational Paradigms* **3526** (2006), p. 263–274. 8, 136

[KK] C. KIRCHNER et H. KIRCHNER – « Rewriting solving proving », Accessible `http ://www.loria.fr/∼ckirchne`. 14

[KL80] S. KAMIN et J-J LÉVY – « Attempts for generalising the recursive path orderings. », Tech. report, Univerity of Illinois, Urbana, 1980, Unpublished note.
http ://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html. 12, 26

[Klo92] J.W. KLOP – « Term rewriting systems », *Handbook of Logic in Computer Science*, vol. 2, Oxford University Press, 1992, p. 1–116. 14, 49

[KN85] M. S. KRISHNAMOORTHY et P. NARENDRAN – « On recursive path ordering », *Theoretical Computer Science* **40** (1985), no. 2-3, p. 323–328. 27

[KN04] L. KRISTIANSEN et K.H. NIGGL – « On the computational complexity of imperative programming languages », *Theoretical Computer Science* **318** (2004), no. 1-2, p. 139–161. 8

[KO92] M. KURIHARA et A. OHUCHI – « Modularity of simple termination of term rewriting systems with shared constructors », *Theoretical Computer Science* **103** (1992), p. 273–282. 49, 51

[Laf04] Y. LAFONT – « Soft linear logic and polynomial time », *Theoretical Computer Science* **318** (2004), no. 1-2, p. 163–180. 8, 162

[Lan79] D.S. Lankford – « On proving term rewriting systems are noetherien », Tech. report, 1979. 8, 11, 26, 68

[Lei94] D. Leivant – « Predicative recurrence and computational complexity I: Word recurrence and poly-time », Feasible Mathematics II, Birkhäuser, 1994, p. 320–343. 8, 11

[Lei98] — , « A characterization of NC by tree recurrence. », *FOCS'98*, 1998, p. 716–724. 114

[LJBA01] C. S. Lee, N. D. Jones et A. M. Ben-Amram – « The size-change principle for program termination », *POPL'01*, vol. 28, 2001, p. 81–92. 74, 77, 107, 109

[LM93] D. Leivant et J.-Y. Marion – « Lambda calculus characterizations of poly-time », *Fundamenta Informaticae* **19** (1993), no. 1,2, p. 167,184. 8, 11

[LM00] D. Leivant et J.-Y. Marion – « A characterization of alternating log time by ramified recurrence », *Theoretical Computer Science* **236** (2000), no. 1-2, p. 192–208. 113, 121

[LS07] L. Liquori et A. Spiwack – « Feathertrait : A modest extension of featherweight java », *ACM Transaction on Programming Languages and Systems* (2007), Accepted, to appear. 159

[Luc05] S. Lucas – « Polynomials over the reals in proofs of termination : from theory to practice », *RAIRO Theoretical Informatics and Applications* **39** (2005), no. 3, p. 547–586. 130

[Mar00] J.-Y. Marion – « Complexité implicite des calculs, de la théorie à la pratique », Habilitation à diriger les recherches, Université Nancy 2, 2000. 8, 12

[Mar03] — , « Analysing the implicit complexity of programs », *Information and Computation* **183** (2003), p. 2–18. 8, 11, 74

[Mid90] A. Middeldorp – « Modular properties of term rewriting systems », Thèse, Vrije Universiteit te Amsterdam, 1990. 49

[Min06] A. Miné – « Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics », *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers and tool support for embedded systems*, ACM Press New York, NY, USA, 2006, p. 54–63. 141

[MM00] J.-Y. Marion et J.-Y. Moyen – « Efficient first order functional program interpreter with time bound certifications », *LPAR 2000*, Lecture Notes in

Computer Science, vol. 1955, Springer, 2000, p. 25–42. 8, 11, 12, 24, 28, 30, 74

[MN70] Z. Manna et S. Ness – « On the termination of Markov algorithms », *HICSS 3*, 1970, p. 789–792. 11, 26, 68

[Moy03] J.-Y. Moyen – « Analyse de la complexité et transformation de programmes », Thèse d'université, Nancy 2, Dec 2003. 11

[MP06a] J.-Y. Marion et R. Péchoux – « Quasi-friendly sup-interpretations », *Workshop on Logic and Complexity in Computer Science, LCC 2006*, 2006, http://hal.inria.fr. 9

[MP06b] — , « Resource analysis by sup-interpretation », *FLOPS 2006*, Lecture Notes in Computer Science, vol. 3945, Springer, 2006, p. 163–176. 8, 9, 73, 74, 82

[MP07a] — , « Resource control of object-oriented programs », *Workshop on Logic and Complexity in Computer Science, LCC 2007*, 2007, http://hal.inria.fr. 10

[MP07b] J.Y. Marion et R. Péchoux – « A characterization of polynomial complexity classes using dependency pairs », 2007, Accessible on http://hal.inria.fr. 10, 130

[MP07c] — , « Sup-interpretations, a semantic method for static analysis of program resources », *Transactions on Computational Logic* (2007), Submitted. 8, 9, 73

[Nig98] K.-H. Niggl – « The $\mu$-measure as a tool for classifying computational complexity », *The bulletin of symbolic logic* **4** (1998), no. 1, p. 100–101. 8, 12

[Nig00] K.H. Niggl – « The-measure as a tool for classifying computational complexity », *Archive for Mathematical Logic* **39** (2000), p. 515–539. 8

[NW06] K.H. Niggl et H. Wunderlich – « Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs », *SIAM Journal on Computing* **35** (2006), p. 1122. 8, 136, 155

[Ohl02] E. Ohlebusch – *Advanced Topics in Term Rewriting*, Springer, 2002. 49

[Pap94] C. H. Papadimitriou – *Computational complexity*, Addison-Wesley, 1994. 6, 7, 30, 33, 111

[Péc05] R. Péchoux – « Synthèse de quasi-interprétations », *Mémoire de* DEA. *Institut National Polytechnique de Lorraine* (2005). 9, 42

[Rao95] M.R.K. Krishna Rao – « Modular proofs of completeness of hierarchical term rewriting systems », *Theoretical Computer Science* **151** (1995), p. 487–512. 52

[Rey98] J.C. Reynolds – « Definitional Interpreters for Higher-Order Programming Languages », *Higher-Order and Symbolic Computation* **11** (1998), no. 4, p. 363–397. 69

[Ruz81] W. Ruzzo – « On uniform circuit complexity. », *Journal of Computer System Science* **22** (1981), no. 3, p. 365–383. 110, 112, 113, 119

[Sav70] W. J. Savitch – « Relationship between nondeterministic and deterministic tape classes », *Journal of Computer System Science* **4** (1970), p. 177–192. 107

[Saz80] V.Y. Sazonov – « Polynomial Computability and Recursivity in Finite Domains », *Elektronische Informationsverarbeitung und Kybernetik* **16** (1980), no. 7, p. 319–323. 8

[Sip96] M. Sipser – « Introduction to the Theory of Computation PWS Pub », 1996. 22

[Tar51] A. Tarski – *A decision method for elementary algebra and geometry, 2nd ed*, University of California Press, 1951. 35, 37

[TG03] R. Thiemann et J. Giesl – « Size-change termination for term rewriting », *RTA 2003* (Valencia, Spain), Lecture Notes in Computer Science, Springer, 2003. 107

[Toy87a] Y. Toyama – « Counterexamples for the direct sum of term rewriting systems », *Information Processing Letters* **25** (1987), p. 141–143. 49

[Toy87b] — , « On the church-rosser property for the direct sum of term rewriting systems », *Journal of the ACM* **34** (1987), no. 1, p. 128–143. 49

[Tur36] A. M. Turing – « On computable numbers with an application to the entscheidungsproblem », *Proc. London Mathematical Society* **42** (1936), no. 2, p. 230–265. 6, 22

[Wei95] A. Weiermann – « Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths », *Theoretical Computer Science* **139** (1995), p. 335–362. 27

# Table des figures


I.1 Syntaxe d'un programme **p** . . . . . . . . 14
I.2 Sémantique par appel par valeur d'un programme **p** . . . . . . . . 15
I.3 Arbre des appels correspondant à l'évaluation de $\mathtt{q}(\mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{0})))), \mathbf{S}(\mathbf{S}(\mathbf{0})))$ 18
I.4 Définition de $\prec_{rpo}$ . . . . . . . . 27
I.5 Évaluation d'un programme à l'aide d'un interpréteur avec cache . . . . . 29
I.6 Tableau récapitulatif des valeurs de la quasi-interprétation $(\!|\mathtt{f}_i|\!)$ . . . . . . 46
I.7 Unions de deux programmes $\langle Var_1, Cns_1, Fct_1, R_1 \rangle$ et $\langle Var_2, Cns_2, Fct_2, R_2 \rangle$ 50

II.1 Syntaxe des expressions avec opérateurs . . . . . . . . 76
II.2 Sémantique par appel par valeur d'un opérateur . . . . . . . . 77
II.3 Génération de contraintes sur les projecteurs . . . . . . . . 98
II.4 SCP graphe du programme `minus` . . . . . . . . 108
II.5 $\theta$-SCP graphe du programme `q` . . . . . . . . 110

III.1 Syntaxe des classes . . . . . . . . 138
III.2 Sémantique opérationnelle des expressions . . . . . . . . 140
III.3 Sémantique opérationnelle des commandes . . . . . . . . 141

# Index


Algèbre de fonctions, 7
Alternance, 60
Analyse statique, 8
Arbre des appels, 17
Assignation, 18
Assignation
  additive, 19
  objet, 142
  polynomiale, 19
Attribut, 138

Bernard C., 4
Besoins humains, 3

Calculabilité, 23
Call-tree, 17
Carnap R., 12
Classe, 138
Classe
  de complexité, 6
  principale, 138
Codage de Huffman, 87
Commande, 138
Commande
  de type loop, 147
  de type while, 147
  externe, 147
Complétude intensionnelle, 12
Concatenation Recursion on Notation (CRN), 123
Configuration d'une Machine de Turing, 23
Contexte, 16
Contraintes sur les projecteurs, 98
Critère
  à appels récursifs bornés, 91
  à appels récursifs strictement bornés, 105
  arborescent, 114
  fraternel, 148
  quasi-amical, 82
  quasi-amical modulo projection, 98
  strictement fraternel, 157
  sympathique, 117

Définition , 14
Défonctionnalisation, 69
Dénotation, 11
Degré
  max-degré, 38
  $\times$-degré, 38
  $+$-degré, 41

État, 17
Expression, 138
Expression , 14
  activée, 16
  maximale, 16
Extension
  de quasi-interprétation, 62

lexicographique, 26
multi-ensemble, 27
produit, 26
Extensionnalité, 11

Forme canonique d'une fraternité, 78
FPSPACE, 23
FPTIME, 23
Fraternité, 77
Frege G., 11

Intensionnalité, 11
Interprétation d'espace polynomial, 130

Kennedy J.F., 161

Lemme fondamental, 23
Lewis Trondheim, 135

Méthode, 138
Méthode
fraternelle, 153
Machine de Turing, 22
Machine-indépendance, 7
Magasin, 140
Maslow A., 4
**Max-Plus**, 18
**Max-Poly**, 19
**(k,d)-Max-Poly**, 38
Modèles de calcul, 6
Motif , 14

Non-déterminisme, 6

Objet, 140
Opérateur, 76
Opérateur
admissible en espace, 155
admissible en temps, 157

*k*-opérateur, 120

Paire de dépendance, 103
Parallel Register Machine (PRM), 32
Poids, 82
Poids
d'une méthode, 153
faible, 114
objet, 147
Polynôme honnête, 58
Précédence, 17
Problème
de décision, 6
de la synthèse, 35
de séparation, 7
Programme
à appels récursifs bornés, 91
à appels récursifs strictement bornés, 105
arborescent, 114
fraternel, 148
quasi-amical, 82
quasi-amical modulo projection, 98
strictement fraternel, 157
sympathique, 117
Projecteur, 97
Propriété modulaire, 49
Pyramide des besoins, 4

Quasi-interprétation, 19
Quasi-interprétation
-s additivement préservatrices, 59
-s préservatrices, 58
irrationnelle, 36
Quasi-ordre, 104
Quasi-ordre
bien-fondé, 104

clos par substitution, 104
monotone, 104
Queneau R., 73
Quicksort, 99

Recursive Path Ordering (RPO), 26
Register Machine (RM), 29
Ressources informatiques, 5

SCP graphes, 108
$\theta$ SCP graphes, 109
SCP multi-chemins, 108
Sens, 12
Size-change principle (SCP), 107
Stack frame, 17
Streams, 94
Substitution, 15
Sup-interprétation, 79
Sup-interprétation
de projecteur, 97
objet, 144

Taille, 19, 142
Termes clos, 15
Théorie de la complexité, 5
Thucydide, 11
Transition, 17

Union
à constructeurs partagés, 52
disjointe, 50
hiérarchique, 56
stratifiée, 60

Valeur, 15

Weak bounded Recursion on Notation (WBRN), 124

INSTITUT NATIONAL
POLYTECHNIQUE
DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE LORRAINE

**o0o**

VU LES RAPPORTS ETABLIS PAR :
**Monsieur Roberto AMADIO, Professeur, Université Paris 7, Paris**

**Monsieur Neil JONES, Professeur, University of Copenhagen, DENMARK**

Le Président de l'Institut National Polytechnique de Lorraine, autorise :

**Monsieur PÉCHOUX Romain**

NANCY BRABOIS
2, AVENUE DE LA
FORET-DE-HAYE
BOITE POSTALE 3
F - 54501
VANDŒUVRE CEDEX

à soutenir devant un jury de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

**"Analyse de la complexité des programmes par interprétation sémantique"**

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Spécialité : « **Informatique** »

Fait à Vandoeuvre, le 29 octobre 2007
Le Président de l'I.N.P.L.,
F. LAURENT

TEL. 33/03.83.59.59.59
FAX. 33/03.83.59.59.55

# Résumé

Il existe de nombreuses approches développées par la communauté *Implicit Computational Complexity* (ICC) permettant d'analyser les ressources nécessaires à la bonne exécution des algorithmes. Dans cette thèse, nous nous intéressons plus particulièrement au contrôle des ressources à l'aide d'interprétations sémantiques. Après avoir rappelé brièvement la notion de quasi-interprétation ainsi que les différentes propriétés et caractérisations qui en découlent, nous présentons les différentes avancées obtenues dans l'étude de cet outil : nous étudions le problème de la synthèse qui consiste à trouver une quasi-interprétation pour un programme donné, puis, nous abordons la question de la modularité des quasi-interprétations. La modularité permet de diminuer la complexité de la procédure de synthèse et de capturer un plus grand nombre d'algorithmes. Après avoir mentionné différentes extensions des quasi-interprétations à des langages de programmation réactif, bytecode ou d'ordre supérieur, nous introduisons la sup-interprétation. Cette notion généralise la quasi-interprétation et est utilisée dans des critères de contrôle des ressources afin d'étudier la complexité d'un plus grand nombre d'algorithmes dont des algorithmes sur des données infinies ou des algorithmes de type « diviser pour régner ». Nous combinons cette notion à différents critères de terminaison comme les ordres RPO, les paires de dépendance ou le size-change principle et nous la comparons à la notion de quasi-interprétation. En outre, après avoir caractérisé des petites classes de complexité parallèles, nous donnons quelques heuristiques permettant de synthétiser des sup-interprétations sans la propriété sous-terme, c'est à dire des sup-interprétations qui ne sont pas des quasi-interprétations. Enfin, dans un dernier chapitre, nous adaptons les sup-interprétations à des langages orientés-objet, obtenant ainsi différents critères pour contrôler les ressources d'un programme objet et de ses méthodes.

**Mots clefs :** complexité implicite, analyse statique, contrôle des ressources, interprétations

# Abstract

There are several approaches developed by the *Implicit Computational Complexity* (ICC) community which try to analyze and control program resources. In this document, we focus our study on the resource control with the help of semantics interpretations. After introducing the notion of quasi-interpretation together with its distinct properties and characterizations, we show the results obtained in the study of such a tool: We study the synthesis problem which consists in finding a quasi-interpretation for a given program and we tackle the issue of quasi-interpretation modularity. Modularity allows to decrease the complexity of the synthesis procedure and to capture more algorithms. We present several extensions of quasi-interpretations to reactive programming, bytecode verification or higher-order programming. Afterwards, we introduce the notion of sup-interpretation. This notion strictly generalizes the one of quasi-interpretation and is used in distinct criteria in order to control the resources of more algorithms, including algorithms over infinite data and algorithms using a « divide and conquer » strategy. We combine sup-interpretations with distinct termination criteria, such as RPO orderings, dependency pairs or size-change principle, and we compare them to the notion of quasi-interpretation. Using the notion of sup-interpretation, we characterize small parallel complexity classes. We provide some heuristics for the sup-interpretation synthesis: we manage to synthesize sup-interpretations without the subterm property, that is, sup-interpretations which are not quasi-interpretations. Finally, we extend sup-interpretations to object-oriented programs, thus obtaining distinct criteria for resource control of object-oriented programs and their methods.

**Keywords:** implicit computational complexity, static analysis, resource control, interpretations