Université de Montréal

Complexité des relations sémantiques dans les systèmes de programmation

par

Yves Marcoux

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

Juin 1991
© Yves Marcoux, 1991

Cette thèse a été publiée comme Document de travail #214 au
Département d'informatique et de recherche opérationnelle de l'Université
de Montréal en juin 1992.

Des corrections mineures ont été apportées le 2017-01-25.

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée :

Complexité des relations sémantiques dans les systèmes de programmation

présentée par :

Yves Marcoux

a été évaluée par un jury composé des personnes suivantes :

Monsieur Claude Christen, président-rapporteur
Monsieur Pierre McKenzie, directeur de recherche
Monsieur Gilles Brassard, codirecteur de recherche
Monsieur Geňa Hahn, membre du jury
Monsieur Kenneth W. Regan, examinateur externe (SUNY Buffalo)

Thèse acceptée le : 29 août 1991

Sommaire

La plupart des propriétés de programmation étudiées dans la littérature correspondent à l'existence de transformations effectives de programmes établissant un certain "rapport sémantique" entre programmes originels et transformés. Riccardi [Ric80] a introduit le cadre formel des *structures de contrôle*, qui permet de définir une classe de "rapports sémantiques". Cette classe est très vaste et englobe la plupart des propriétés de programmation couramment utilisées ou étudiées. Case et Royer [Ric80, Roy87] ont obtenu un théorème de *complétude expressive* : les systèmes de programmation *acceptables* sont précisément ceux dotés de *toutes* les propriétés exprimables dans le formalisme de Riccardi. Un tel résultat nous renseigne sur la *similitude* des systèmes de programmation acceptables, et sur leur *versatilité*, en l'occurrence, la *diversité* des techniques de programmation qui y sont utilisables.

Nous montrons qu'une certaine technique de programmation *naturelle*, la *pseudo-inversion*, n'est pas exprimable dans le formalisme de Riccardi. Nous proposons deux extensions de la structure de contrôle : la *relation sémantique* et la *relation vérifiable par énumération*. La première fournit un cadre général pour l'étude des "rapports sémantiques". La seconde, plus restreinte, englobe malgré tout la pseudo-inversion et donne lieu à un résultat de complétude expressive strictement plus fort que celui de Case et Royer. Nous obtenons deux nouvelles caractérisations des structures de contrôle *extensionnelles*, qui correspondent aux "rapports sémantiques" exprimables en sémantique dénotationnelle et dans la théorie des schèmes de programmes [Roy87].

Trois relations sémantiques importantes sont s-1-1, comp et prog (ce sont aussi des structures de contrôle). Tout système de programmation exécutable (i.e., dont on peut effectivement interpréter les programmes) doté d'une des propriétés correspondantes est nécessairement acceptable. Comme techniques de programmation, elles correspondent respectivement à la spécialisation d'un programme pour une valeur précise d'un paramètre, à la composition fonctionnelle de programmes et à la traduction (compilation) de programmes d'un langage standard.

On dit qu'une relation sémantique en *garantit* une autre ssi le fait de maîtriser

la première (comme technique de programmation) implique la maîtrise de la seconde. Nous complétons l'étude, entreprise dans [Ric82], des garanties existant entre s-1-1, comp et prog dans les systèmes de programmation qui ne sont pas nécessairement exécutables. Nous répondons entre autres à une question ouverte.

Nous nous intéressons aux *complexités relatives*, dans un même système de programmation, des transformations de programmes correspondant à deux relations sémantiques dont l'une garantit l'autre. Nous montrons que s-1-1 garantit efficacement une vaste classe de relations vérifiables par énumération, améliorant ainsi certains résultats de Royer, et nous identifions une classe de telles relations comme représentante raisonnable de la "tâche de programmer" dans un système de programmation. Nous réfutons une intuition exprimée à deux reprises dans la littérature [MWY78, Roy87], selon laquelle la garantie de s-1-1 par comp est nécessairement *très* inefficace. Nous présentons une réalisation relativement efficace et *optimale* de cette garantie. La réalisation n'étant que "relativement" efficace, nous concluons que s-1-1 est plus fondamentale que comp, même si la "différence" est moindre que prévue.

Nous étudions le phénomène de *non-constructivité* des garanties, de même que certains problèmes de recherche et de décision sur les systèmes de programmation. Nous démontrons la non-constructivité de la garantie de prog par s-1-1, et la constructivité de la garantie par prog d'une vaste classe de relations vérifiables par énumération. Ces résultats suggèrent (i) qu'un compilateur pour un langage standard constitue une information *nécessaire et suffisante* pour programmer dans un système de programmation acceptable dont on connaît un interprète, et (ii) qu'on ne peut déduire cette information de l'interprète seulement. La connaissance d'un interprète d'un langage (sa *sémantique opérationnelle*) ne serait donc pas toujours suffisante pour programmer : un "guide du programmeur", donnant explicitement ou implicitement un compilateur pour un langage standard, serait en général nécessaire.

Mots clés : Systèmes de programmation, Structures de contrôle, Sémantique, Calculabilité, Complexité.

Table des matières

Identification du jury ii

Sommaire iii

Mots clés v

Liste des abréviations x

Remerciements xii

1 Motivation, aperçu, préliminaires 1

1.1 Introduction informelle et aperçu 1
1.2 Travaux connexes 13
1.3 Préliminaires 15
1.3.1 Généralités 15
1.3.2 Fonctions, relations 16
1.3.3 Notations asymptotiques 18
1.3.4 Modèle de calcul, complexité 19
1.3.5 Calculabilité I 22
1.3.6 Systèmes de programmation 24
1.3.7 Calculabilité II, opérateurs récursifs 29
1.3.8 Séquences finies d'entiers 32
1.3.9 Autres conventions de notation 32

2 Structures de contrôle, relations sémantiques, relations vérifiables par énumération **33**
2.1 Structures de contrôle . . . 34
2.2 Relations sémantiques . . . 39
2.3 Relations vérifiables par énumération . . . 43
2.4 Quelques RVE . . . 48
2.5 Extensionnalité . . . 51
2.6 Relations de garantie . . . 70
2.7 Similitude et versatilité des SP acceptables . . . 72
2.8 Discussion . . . 78

3 Complexité des instances : s-1-1 contre composition **80**
3.1 Introduction . . . 80
3.2 Les interrelations s-1-1 $\models_{em}$ CC-RVE . . . 80
3.2.1 Quelques résultats positifs . . . 80
3.2.2 Un résultat négatif . . . 89
3.2.3 Remarques . . . 91
3.3 L'interrelation comp $\models_m$ s-1-1 . . . 92
3.3.1 La borne supérieure . . . 93
3.3.2 La borne inférieure . . . 100
3.3.3 Remarques concernant s-1-1 contre comp . . . 110

4 Relations sémantiques dans les systèmes de programmation maximaux **117**
4.1 Pré- et post-compositions hybrides . . . 118
4.2 pr-comp-h contre comp, s-1-1 et prog . . . 122
4.3 po-comp-h contre comp, pr-comp-h, s-1-1 et prog . . . 125
4.4 prog contre les RS de Myhill-Shepherdson . . . 132
4.5 Discussion . . . 135

5 Problèmes de décision et de recherche sur les systèmes de programmation **138**
5.1 Quelques problèmes de décision . . . 140
5.2 Constructivité de RVE à partir de prog . . . 147
5.3 Difficulté d'inférer une fonction de programmation . . . 149
5.4 Non-constructivité de s-1-1 $\models_{em}$ prog . . . 153

Conclusion **158**

Bibliographie **161**

Index des symboles **164**

Index des définitions **165**

Table des figures

4.1 Sommaire des résultats des Sections 4.1, 4.2 et 4.3. 119

Liste des abréviations

r.é. : récursivement énumérable
réc. : récursive
RS : relation sémantique (Définition 2.9)
RVE : relation vérifiable par énumération (Définition 2.15)
SP : système de programmation (Définition 1.2)
ssi : si et seulement si

À mes parents,
Denise,
Isabelle et Alain.

Remerciements

Je désire d'abord remercier mon directeur de recherche, Pierre McKenzie, et mon codirecteur, Gilles Brassard, pour la confiance qu'ils m'ont témoignée et leurs encouragements répétés. Par delà les discussions que j'ai eues avec eux et les conseils qu'ils m'ont donnés, sans lesquels ce travail eût été impossible, ils ont été pour moi des modèles.

Je tiens à remercier MM. Claude Christen et Geňa Hahn d'avoir accepté d'être membres du jury d'évaluation de cette thèse, et M. Kenneth Regan, de SUNY Buffalo, d'avoir accepté d'agir à titre d'examinateur externe. Ils me font tous trois un grand honneur.

J'exprime une profonde gratitude à Jim Royer, dont la thèse de doctorat [Roy87] est en quelque sorte à la base de mes travaux et a été pour moi une source constante d'inspiration et un exemple. Le nombre de références que je fais à sa thèse témoigne de l'ampleur de ma dette intellectuelle envers lui. Jim m'a suggéré directement le sujet de la Section 3.3.1, et bien d'autres sujets me sont venus à l'esprit à la lecture de sa thèse. Je le remercie également pour de nombreux et fructueux conseils, discussions et encouragements.

Je suis extrêmement reconnaissant à John Case pour l'intérêt qu'il a porté à mes travaux, et sa grande disponibilité : c'est avec un plaisir toujours vif que je me rappelle les trois jours (tout en discussions scientifiques, philosophiques, psychologiques et autres) passés en sa compagnie à Rochester à l'été 88. Il m'a à cette occasion donné de nombreux conseils et suggéré le sujet de la Section 3.3.2 de même que l'idée (devenue le Théorème 5.18) de généraliser un résultat aux fonctions partielles récursives dans le problème d'arrêt. J'ai profité à plusieurs autres reprises de sa disponibilité. Je tiens particulièrement à souligner que John a grandement aidé mes directeurs de recherche dans l'élaboration d'une liste d'examinateurs externes potentiels.

Jim et John ont tous deux pris le temps de nous rencontrer, mes directeurs de recherche et moi, pour parler de mes travaux à l'occasion de visites à Montréal au printemps 89.

Sur une note plus technique, je remercie Mark Fulk pour une preuve d'un résultat précurseur au Théorème 5.18, Geňa Hahn pour une simplification de ma preuve originelle du Lemme 3.21, et Jim Royer pour l'idée d'utiliser les fonctions constantes pour obtenir une borne inférieure pour s-1-1 (idée utilisée dans la preuve du Théorème 3.18). Je remercie également Mark Fulk, Carl Smith et Paul Young pour des discussions fructueuses.

Je salue la petite "gang" du T-533, particulièrement Hervé Caussinus, Vahé Kassardjian et François Lemieux, et la remercie pour une atmosphère de travail agréable et stimulante. Je remercie aussi mon ami l'écrivain Denys Gagnon pour les nombreuses consultations grammaticales gratuites qu'il m'a accordées, et combien de cafés réconfortants chez Van Houtte !

Finalement, je remercie mon extraordinaire épouse et amie Denise, pour son amour, sa compréhension, ses encouragements ; tout ce qu'elle m'accorde depuis déjà (eh oui, déjà !) si longtemps. Il m'aurait été impossible de mener à bien un tel projet sans cette compagne exceptionnelle.

Je remercie les personnes et organismes suivants pour le support financier qu'ils m'ont fourni : le FCAR, la Faculté des études supérieures de l'Université de Montréal et les Professeurs Pierre McKenzie et Gilles Brassard (fonds provenant de subventions CRSNG).

Chapitre 1

Motivation, aperçu, préliminaires

1.1 Introduction informelle et aperçu

Ce travail se veut une modeste contribution à l'avancement de l'étude systématique des propriétés de programmation des systèmes de programmation entreprise par Riccardi et Case au début des années 80 [Ric80, Ric81, Ric82] et poursuivie quelques années plus tard par Royer [Roy87].

Le modèle des *systèmes de programmation* a été introduit par Rogers en 1958 pour étudier les "formalismes de description d'algorithmes" [Rog58]. Informellement, un système de programmation (abrégé *SP*) est un langage de programmation dans lequel les seuls éléments syntaxiques dotés d'une sémantique sont les *programmes complets*. Les notions *d'expression*, *d'énoncé*, de *bloc*, de *segments de programmes*, etc. sont inexistantes. Les programmes (toujours complets) sont des entiers et *chaque* entier est un programme. La sémantique d'un programme (ce qui est "calculé" par ce programme) est toujours une fonction partielle récursive de N dans N. Toutes les données (entrées et sorties) sont donc, comme les programmes, des entiers. Si un SP a au moins un programme pour chaque fonction partielle récursive, on dit qu'il est *maximal*.

Intuitivement, une propriété de programmation est une propriété des SP qui

nous dit quelque-chose sur la "tâche de programmer" dans les SP qui en sont dotés. Un exemple extrêmement simple de propriété de programmation est le fait qu'un SP possède un programme pour la fonction constante 0. Si un SP a cette propriété, alors on sait qu'on peut, dans ce SP, "écrire un programme pour la fonction constante 0". Ce fait constitue une information, si petite soit-elle, sur la tâche de programmer dans ce SP. Un exemple peut-être un peu plus intéressant de propriété de programmation est le fait qu'il soit possible dans un SP de trouver *uniformément*, à partir de n'importe quelle paire de programmes, un troisième programme calculant la composition fonctionnelle des fonctions partielles calculées par les deux programmes de départ. Un autre exemple est le fait qu'on puisse uniformément traduire (i.e., *compiler*) des descriptions de machines de Turing en programmes *équivalents* dans le système de programmation considéré.

Les propriétés de programmation des SP s'opposent implicitement et intuitivement à leurs propriétés *d'exécution*. Une propriété d'exécution serait informellement une propriété qui nous dit quelque-chose sur la tâche d'exécuter, ou interpréter, les programmes d'un SP. Par exemple, le fait que la tâche d'exécuter un programme sur une entrée requière un temps exponentiel en la longueur du programme (peu importe l'interprète utilisé) serait une propriété d'exécution (peu enviable) de certains systèmes de programmation. La seule propriété d'exécution qui soit couramment utilisée, et certainement la seule à laquelle nous ferons appel dans ce travail, est *l'exécutabilité*, qui correspond au fait que la tâche d'exécuter les programmes est *récursivement faisable*. Un SP doté de cette propriété est dit *exécutable* [Ric80] ou *interprétable*.

Les propriétés de programmation ont le plus souvent été étudiées individuellement et sporadiquement, sans que leur nature de propriété de programmation ne soit soulignée, et parfois dans le contexte de SP spécifiques (surtout avant [Rog58]). Rogers, dès [Rog58], a étudié la propriété de *substitution effective* (aussi appelée propriété *s-1-1*, d'un cas particulier du théorème *s-m-n* de Kleene, reproduit ici comme Théorème 1.9), qui correspond informellement au fait que d'un programme à deux arguments (conceptuellement) et d'une donnée (i.e., d'un entier), l'on puisse *uniformément* construire un autre programme, à un seul argument, qui "fait la même chose" que le programme de départ, mais avec un premier argument fixe égal à la donnée de départ. Intuitivement : dans un SP doté de cette propriété, on peut uniformément substituer une constante à un paramètre formel. Rogers a aussi étudié des propriétés de *traductibilité* (ou *compilabilité*) effective d'un SP vers un autre. Machtey, Winklmann et Young ont introduit et étudié la propriété de *com-*

position effective, que nous avons donnée en exemple plus haut [MWY78].

Dans [Ric80], Riccardi a introduit un cadre formel qui permet de définir collectivement des *classes* de propriétés de programmation, et a alors commencé une étude systématique de ces propriétés. Le formalisme de Riccardi est très général, et il est difficile de donner intuitivement un aperçu de son étendue. En première approximation, cependant, nous décrivons *un* genre de propriétés qu'il permet d'exprimer.

Supposons que l'on définisse une *transformation sémantique*, i.e., une transformation qui à toute sémantique de programme possible associe une sémantique "transformée" ou "image". Dans le contexte des SP, une transformation sémantique est une application de l'ensemble des fonctions partielles récursives dans lui-même. Supposons donc que l'on définisse une telle transformation Φ. Supposons maintenant que dans un certain SP, il soit possible, étant donné n'importe quel programme, de trouver *uniformément* un autre programme calculant l'image par Φ de la fonction partielle calculée par le programme de départ. Clairement, ce fait constitue une propriété de programmation du SP en question : il nous dit que la *technique de programmation* consistant à passer d'un programme pour une certaine fonction partielle à un autre programme pour la *transformée* par Φ de cette fonction partielle est *utilisable* dans ce système de programmation. Si Φ est n'importe quel *opérateur récursif* (intuitivement, une application "effective" de l'ensemble des fonctions partielles récursives dans lui-même), alors la propriété de programmation correspondante est exprimable dans le formalisme de Riccardi. (Dans le langage que nous introduirons plus loin, la classe de propriétés de programmation que nous venons de décrire correspondra aux *relations sémantiques de Myhill-Shepherdson*, Définition 2.54.)

Comme extensions du genre de propriétés que nous venons de présenter, *aussi* exprimables dans le formalisme de Riccardi, mentionnons entre autres les propriétés de programmation correspondant à des transformations sémantiques effectives à plus d'un argument (la composition effective est un exemple d'une telle propriété), et celles correspondant à des transformations sémantiques effectives dont certains arguments sont non pas des "sémantiques de programme", mais simplement des données (la substitution effective est un exemple d'une telle propriété).[1] Cette première approximation du for-

1. L'introduction de ce dernier genre de transformations sémantiques dans le formalisme de Riccardi est nécessaire à l'expression de certaines propriétés intéressantes en raison du fait que seuls les programmes complets ont une sémantique dans les systèmes

malisme de Riccardi nous suffira pour l'instant. (Dans le langage que nous introduirons plus loin, la classe de propriétés de programmation que nous venons de décrire correspondra aux *relations sémantiques extensionnelles sans récursion au sens de Royer*, Définition 2.33.) Mentionnons encore que l'objet formel de base dans le formalisme de Riccardi est la *structure de contrôle*, et qu'il est ainsi appelé parce que *certaines* des techniques de transformation de programmes qu'on peut exprimer grâce à lui correspondent à l'utilisation de constructions syntaxiques directement disponibles comme primitives dans plusieurs langages. Ainsi, Riccardi définit les structures de contrôle loop, if-then-else, while, etc.

Parlons maintenant de l'intérêt de formalismes pour définir des "classes" de propriétés de programmation. Notons d'abord que le modèle même des SP est motivé par le désir plus ou moins explicite de s'attaquer à certaines questions intuitives sur les langages de programmation [2] en suivant une approche *formelle* (par opposition à *empirique*), avec un modèle suffisamment abstrait pour représenter *l'essence* d'un langage de programmation, et non pas une implantation particulière. Un SP est donc essentiellement un langage de programmation, dans une version facile à manipuler formellement. La discussion qui suit s'exprime plus naturellement en termes intuitifs de langages de programmation qu'en termes de SP. Nous y confondrons donc *langages* et *systèmes* de programmation, chose que nous ferons également dans d'autres passages intuitifs de ce travail.

Une question fondamentale dans l'étude des langages de programmation (que l'approche soit formelle ou non) est : comment les décisions de conception d'un langage de programmation influencent-elles ses propriétés à l'utilisation ? Clairement, l'activité de programmation constitue une large part de l'utilisation d'un langage, et conséquemment, les propriétés de programmation d'un langage constituent une large part de ses "propriétés à l'utilisation". Oublions néanmoins pour le moment la distinction entre différentes "sortes" de propriétés des langages de programmation, et intéressons-nous simplement à la question : comment les décisions de conception d'un langage de programmation influencent-elles ses *propriétés ?*

---

de programmation.

2. Nous incluons dans "langages de programmation" les concepts de "modèle de calcul" et "formalisme de description d'algorithmes". L'activité de "programmation", dans un modèle de calcul, serait peut-être plus justement décrite comme étant "l'élaboration de preuves", puisqu'en général, un algorithme dans un modèle de calcul sert à démontrer un fait de calculabilité ou de complexité.

Très intuitivement, pour s'attaquer à cette question (que l'on suive une approche formelle ou non), on fera "varier" les paramètres de conception d'un langage, et on observera les répercussions de ces variations sur les propriétés du langage. Dans une approche formelle, on peut voir une propriété des langages de programmation comme un sous-ensemble de l'ensemble de tous les langages, i.e., comme une chose très précise. On peut alors aussi définir l'objet "à observer" comme étant l'ensemble de toutes les propriétés dont le langage est doté. Cependant, il n'y a *a priori* aucune raison d'espérer pouvoir *décrire*, et encore moins *interpréter*, cet ensemble de façon "intéressante" ou qui touche l'intuition.

Une solution possible est d'utiliser un formalisme de définition de propriétés de langages comme une "grille de référence" pour nous aider à "jauger" ou "sonder" l'ensemble des propriétés d'un langage. Si le formalisme est tel que les propriétés y sont définies de manière "intéressante" ou "éloquente", nous pourrons tirer de l'information "intéressante" ou "éloquente" sur l'ensemble des propriétés dont un langage est doté en étudiant le recoupement entre cet ensemble et notre grille de référence.

C'est comme de telles grilles de référence que nous voyons les différents formalismes d'expression de propriétés des systèmes de programmation, y compris celui de Riccardi. [3] Nous décrivons ces formalismes comme des cadres de définition de propriétés *de programmation*, parce qu'on y perçoit nettement une préoccupation pour l'expression de telles propriétés. Cependant, nous avons acquis la conviction, après avoir étudié ces formalismes d'assez près, que tout formalisme d'expression de propriétés de programmation (pas seulement celui de Riccardi) englobe nécessairement, s'il est assez général, des propriétés qui ne se rapportent que bien indirectement à la "tâche de programmer". Nous conservons malgré tout la distinction entre "propriétés de programmation" et les "autres propriétés" d'un langage, mais cette distinction sera fondamentalement subjective et informelle. Nous respectons en cela Machtey, Winklmann et Young, qui ont introduit la locution "propriété de programmation" comme une expression informelle.

Chaque formalisme représente un "compromis" entre la possibilité d'exprimer

3. Nous utilisons le pluriel parce que Royer a introduit des variantes du formalisme de Riccardi, et que nous introduirons nous-mêmes un nouveau formalisme et quelques variantes. Également, certaines disciplines dans lesquelles intervient une notion de "rapport sémantique", comme par exemple la sémantique dénotationnelle [Sto77] et la théorie des *schèmes de programmes* [Gre75], peuvent être considérées comme de tels formalismes.

beaucoup de propriétés (voire même, toutes), et la quantité "d'information intuitive" contenue dans les définitions des propriétés. Intuitivement, plus un formalisme permet d'exprimer de propriétés différentes, moins l'expression de ces propriétés peut avoir de signification intuitive. Un formalisme "trop puissant", i.e., permettant d'exprimer "trop" de propriétés, bien que constituant une grille de référence très fine, n'aura pas un contenu intuitif suffisant pour être vraiment intéressant ; un formalisme trop "faible" fournira une grille trop grossière malgré qu'il puisse posséder un contenu intuitif très grand. Le formalisme des *relations sémantiques*, dont nous parlerons sous peu, constitue un exemple de formalisme "trop puissant" ; il permet d'exprimer *toutes* les propriétés des SP.

Notre première contribution (notre Chapitre 2) à l'étude des propriétés de programmation dans ce travail pourrait être décrite comme la constatation de la puissance expressive du formalisme de Riccardi et de ses limites. Sur le premier point, nous avons apparemment été le premier à remarquer qu'une propriété intuitivement *très* proche de l'exécutabilité, de même que *l'acceptabilité* des SP, dont la définition (Définition 2.31 et ci-dessous) fait appel directement à l'exécutabilité, sont exprimables dans le formalisme de Riccardi.

Pour parler des limites que nous avons remarquées dans le formalisme de Riccardi, rappelons que les propriétés qu'il permet d'exprimer peuvent être interprétées en termes de *techniques de programmation* (cela était vrai de la première approximation donnée ci-dessus, et le demeure pour le formalisme pleine puissance). Nous avons remarqué que certaines techniques de programmation *naturelles* n'étaient pas exprimables dans le formalisme de Riccardi. Par exemple, nous montrons que la technique de *pseudo-inversion* d'un programme (correspondant à passer uniformément d'un programme pour une fonction partielle à un autre programme pour "l'inverse" de cette fonction partielle, où "l'inverse" est définie de façon sensée pour les fonctions partielles non-bijectives) n'est *pas* exprimable dans le formalisme de Riccardi. La pseudo-inversion est une technique de programmation intuitivement naturelle, avec possiblement des applications pratiques pour la synthèse d'algorithmes à partir de spécifications fonctionnelles (voir § 1.2). En gros, la raison pour laquelle la pseudo-inversion n'est pas exprimable dans le formalisme de Riccardi est qu'en général, une fonction partielle posséde plusieurs pseudo-inverses. La relation entre la sémantique du programme de départ et celle du programme recherché n'est donc pas une transformation, mais bel et bien une *relation*. En général, les techniques de transformation de

programmes où pour un même programme de départ plusieurs programmes transformés *inéquivalents* (i.e., ne calculant pas la même fonction partielle) sont admissibles, ne sont *pas* exprimables dans le formalisme de Riccardi. [4]

Le fait que certaines techniques de programmation *naturelles* ne soient pas exprimables dans le formalisme de Riccardi laisse à penser qu'on pourrait gagner à le raffiner. En effet, étudier l'intersection entre les propriétés d'un système de programmation et celles de Riccardi ne dit *rien* sur certaines propriétés pourtant très naturelles.

Face à cette situation, nous proposons d'élargir le formalisme de Riccardi. Cet élargissement se fait en deux étapes. Tout d'abord, nous effectuons la généralisation la plus grande possible du formalisme, tout en en retenant l'esprit. Cette généralisation est basée sur une catégorie d'objets formels que nous appelons les *relations sémantiques*. En termes de techniques de programmation par transformation de programmes, une relation sémantique (abrégé *RS*) correspond (encore ici, en première approximation) à une relation qui doit exister entre la *sémantique* du programme de départ et celle du programme transformé. Cependant, de façon à obtenir la notion la plus générale possible de technique de programmation, nous n'exigeons aucune espèce de “récursivité” de nos relations sémantiques, contrairement au formalisme de Riccardi qui est basé sur les transformations sémantiques *effectives* (les opérateurs récursifs). Conséquemment, la grille de référence de propriétés de programmation obtenue est extrêmement fine, en fait la plus fine possible. [5] Un corollaire à cette finesse, comme nous avons fait remarquer précédemment, est que le “contenu intuitif” est en général inexistant. En d'autres termes, étant donnée une RS arbitraire, il n'y a *a priori* aucune interprétation intuitive naturelle pour la “technique de programmation” correspondante. La raison pour laquelle nous introduisons la RS n'est donc *pas* pour la proposer comme grille de référence, mais bien pour délimiter jusqu'où on peut aller dans la généralisation du formalisme de Riccardi. En particulier, à l'aide de la RS, nous pouvons maintenant *définir* formellement la pseudo-inversibilité

4. Cette explication n'est pas totalement satisfaisante, car le formalisme de Riccardi, dans sa version pleine puissance, *permet* l'expression de certaines “relations” sémantiques ; simplement, pas celle correspondant à la pseudo-inversion. Notre Corollaire 2.52 indique cependant que l'explication est malgré tout valable dans bien des cas.

5. En fait, les relations sémantiques permettent d'exprimer *toutes* les propriétés des systèmes de programmation, dans le sens que toute classe de systèmes de programmation peut être décrite comme étant la classe des systèmes de programmation dans lesquels la technique de programmation correspondant à une relation sémantique spécifique est utilisable.

(d'une façon qui en fait bien ressortir l'aspect technique de programmation), et énoncer le fait qu'elle n'est pas exprimable dans le formalisme de Riccardi.

Nous définissons ensuite une restriction de la RS que nous appelons la *relation vérifiable par énumération* (abrégé *RVE*). Nous montrons que la classe de propriétés de programmation exprimables via la RVE est strictement plus grande que celle issue du formalisme de Riccardi, et qu'elle contient en particulier la *pseudo-inversibilité*. La RVE étant définie de façon à retenir le même niveau d'intuition que le formalisme de Riccardi, nous avons atteint notre but : une grille de référence strictement plus fine que celle de Riccardi, mais de contenu intuitif aussi grand.

Notre introduction de la RS et de la RVE est complétée par une comparaison plus approfondie avec le formalisme de Riccardi, particulièrement certaines restrictions définies par Royer, comme par exemple les *structures de contrôle extensionnelles* [Roy87] (que nous appelons *relations sémantiques extensionnelles sans récursion au sens de Royer*). Nous mentionnerons sous peu un intérêt de considérer ces restrictions.

Une RS dont la technique de programmation correspondante est utilisable dans un SP donné est dite *récursivement satisfaisable* dans ce système de programmation, intuitivement, parce qu'il existe une transformation *récursive* de programmes telle que programmes de départ et transformé "satisfont" la RS. Une telle transformation récursive est dite *instance effective* de la RS *dans* le SP. On dit que le SP *possède* une instance effective de la RS. Les RS qui sont exprimables dans le formalisme de Riccardi sont appelées *relations sémantiques de Riccardi*.

Une classe *extrêmement* importante de SP est celle des *systèmes de programmation acceptables*. À peu près sans exception, tous les langages de programmation d'usage général rencontrés en pratique correspondent à des SP acceptables. Intuitivement, un SP est acceptable ssi, comme modèle de calcul, il donne lieu à essentiellement la même théorie de la calculabilité que les modèles standard (machine de Turing, RAM, lambda-calcul, etc.) Les SP acceptables ont été définis dans [Rog58] comme suit : un SP est acceptable ssi (i) il est exécutable et (ii) on peut compiler vers lui les programmes d'un des modèles standard. La propriété (ii) a été baptisée *programmabilité* par Riccardi ; elle est exprimable dans son formalisme.

Les termes de la définition formelle de Rogers sont loin de correspondre à la

description intuitive que nous avons donnée des SP acceptables. Pourtant, leur étude de plus en plus approfondie n'a fait que confirmer la validité de cette description intuitive. Peut-on espérer "confirmer" formellement cette validité ? Un pas dans cette direction est d'essayer de montrer que les SP acceptables ont beaucoup de *propriétés* en commun avec les modèles de calcul standard (et donc, entre eux, puisque les modèles standard sont acceptables). Peuvent alors intervenir nos "grilles de référence".

Un autre aspect de l'étude systématique des propriétés de programmation des SP acceptables est de comprendre en quoi ils constituent les "bons" modèles de calcul : on peut estimer la versatilité d'un modèle en démontrant la variété des techniques de programmation (ou de preuve) qui y sont utilisables. Donc, autant pour démontrer que les SP acceptables ont beaucoup de propriétés en commun que pour estimer leur versatilité, nos grilles de référence peuvent servir. On voit bien ici que la finesse de la grille de référence, autant que son niveau d'intuition, influence l'intérêt des réponses obtenues.

Case [Ric80] a déjà obtenu une caractérisation des SP acceptables en termes des propriétés de programmation du formalisme de Riccardi (un théorème de *complétude expressive*). Ce résultat a été légèrement amélioré par Royer [Roy87]. Abstraction faite de certaines conditions techniques sur les propriétés, ce résultat montre que les SP acceptables sont exactement ceux des SP exécutables possédant une instance effective de *chaque* RS de Riccardi. (Notons qu'un corollaire à notre observation qu'une propriété très proche de l'exécutabilité est exprimable dans le formalisme de Riccardi est que le mot "exécutables" peut être éliminé de la phrase précédente.) Nous avons un résultat tout à fait analogue pour les RVE (le Théorème 2.69). Hormis certaines conditions techniques, nous montrons que les SP acceptables sont exactement ceux qui possédent une instance effective de *chaque* RVE. Compte tenu du fait que le formalisme des RVE est strictement plus expressif que celui de Riccardi, ce résultat constitue une réponse *plus précise* aux interrogations exprimées ci-dessus que les résultats de Case et Royer.

Une autre question intéressante concernant les SP acceptables est de savoir quelles propriétés *garantissent* l'acceptabilité d'un SP. Les résultats sur la versatilité des SP acceptables répondent intuitivement à la question : si on a un "bon" langage (un langage acceptable), quelles sont *toutes* les techniques de programmation qu'on peut utiliser ? La présente question demande plutôt : que doit-on mettre *au minimum* dans un langage pour qu'il soit "bon" ? La *définition* même de SP acceptable fournit une telle "garantie minimale"

de l'acceptabilité. Il s'avère qu'un grand nombre d'autres *petits* ensembles de propriétés garantissent l'acceptabilité. Ici, la "petitesse" des ensembles réfère non seulement à leur cardinalité, mais aussi à la "simplicité" de leurs membres. C'est ici que les restrictions de formalismes d'expression de propriétés de programmation ont un intérêt. Intuitivement, une propriété exprimable dans une restriction d'un formalisme est plus "simple" qu'une autre qui n'est exprimable que dans le formalisme non-restreint.

La garantie la plus "petite" de l'acceptabilité que nous connaissions est due à Royer, qui a montré que l'approximation la plus rudimentaire du formalisme de Riccardi que nous avons présentée ci-dessus (les *relations sémantiques de Myhill-Shepherdson*) permet d'exprimer une propriété de programmation qui, à elle seule, garantit l'acceptabilité dans les SP exécutables maximaux. Nous avons cependant ici un résultat négatif (le Théorème 4.12) : toute garantie de l'acceptabilité par une unique RS de Myhill-Shepherdson est nécessairement "faible", dans le sens qu'une telle RS ne peut garantir la *programmabilité* (une des deux propriétés définissant l'acceptabilité), que dans le contexte des SP maximaux *exécutables*. Nous reparlerons sous peu de ce résultat.

Cette discussion de l'acceptabilité nous amène à parler de la notion de "puissance expressive" d'une RS et des interrelations de garantie entre deux RS. On dit qu'une RS rs-a *garantit* une autre RS rs-b ssi pour "tout" SP, le fait de posséder une instance effective de rs-a *implique* la possession d'une instance effective de rs-b. Deux RS sont *équivalentes* ssi elles se garantissent l'une l'autre. On étend cette définition de la façon naturelle aux *ensembles* de RS. La *puissance expressive* d'une RS rs-a est l'ensemble de toutes les RS garanties par rs-a. Cette définition de garantie est compatible avec l'idée intuitive d'une propriété qui en garantit une autre, lorsqu'on voit les RS comme exprimant des propriétés des systèmes de programmation. Un résultat sur la versatilité d'utilisation des SP acceptables nous dit que la propriété d'acceptabilité garantit un *vaste* ensemble de RS ; une garantie de l'acceptabilité nous dit qu'un certain ensemble (le plus souvent, "petit") de RS garantit la propriété d'acceptabilité. Deux RS sont dites *indépendantes* ssi aucune d'entre elles ne garantit l'autre.

Riccardi [Ric80] a observé que les interrelations de garantie entre RS pouvaient varier selon la classe de SP à laquelle réfèrent les mots "*tout* SP" dans la définition de garantie. En particulier, il a montré que certaines interrelations valides dans le contexte des SP exécutables n'étaient *plus* valides

si tous les SP (même non-exécutables) étaient considérés. Évidemment, on peut dire que l'étude des interrelations de garantie dans le contexte des SP arbitraires présente peu d'intérêt, puisque les SP non-exécutables ne peuvent correspondre à aucune forme "réalisable" de langage de programmation. Cependant, l'étude des RS dans le contexte des SP arbitraires révèle mieux certains aspects de leur "puissance combinatoire" pure que leur étude dans le contexte des SP exécutables. En effet, une interprétation intuitive du fait qu'une interrelation de garantie, mettons de rs-b par rs-a, valide pour les SP exécutables, ne soit *pas* valide pour les SP arbitraires est la suivante : dans au moins un SP *exécutable*, toute utilisation de la technique de programmation correspondant à rs-b, basée sur la seule maîtrise de la technique correspondant à rs-a, *exige* la disponibilité d'un interprète du SP. Par contraste, si l'interrelation est valide dans les SP arbitraires, on déduit que dans n'importe quel SP (même les SP *exécutables*) on peut, à partir d'une instance effective de rs-a et peut-être de certains programmes spécifiques, construire une instance effective de rs-b *sans connaître un interprète du langage*. En ce sens, les interrelations de garantie valides *seulement* dans le contexte des SP exécutables sont moins "robustes" que celles valides pour *tous* les SP. Un exemple d'une telle interrelation est la garantie de la RS correspondant à la composition effective par celle correspondant à la substitution effective.

Riccardi a étudié [Ric80, Ric82] les interrelations de garantie entre certaines RS dans le contexte des SP maximaux arbitraires (i.e., pas nécessairement exécutables), et nous poursuivons cette étude (notre Chapitre 4). Les RS étudiées sont celles correspondant aux propriétés de substitution effective (s-1-1) et de composition effective, de même que certaines versions affaiblies de cette dernière. Ces RS sont fondamentales dans l'étude des SP car elles garantissent chacune, à elle seule, l'acceptabilité dans les SP exécutables maximaux. Nous définissons des notions et obtenons des résultats complémentaires à [Ric82], dont la réponse à une question ouverte. Nos résultats sont principalement des résultats d'indépendance. Nous étudions aussi les dépendances entre la programmabilité et les RS de Myhill-Shepherdson. Nous montrons entre autres qu'aucune RS de Myhill-Shepherdson ne peut garantir à elle seule la programmabilité dans les SP maximaux arbitraires (le Théorème 4.12, mentionné précédemment).

Jusqu'ici, nous avons identifié l'*utilisabilité* d'une technique de programmation au fait que la transformation de programmes correspondante soit *uniforme*, i.e., récursive en tant que fonction de N dans N. Mais l'utilisabilité *pratique* d'une technique de programmation est liée autant à la *complexité* de

la transformation de programmes correspondante qu'à sa récursivité. Donc, la versatilité d'un système de programmation est "qualifiée" par la *complexité* d'utiliser chaque technique de programmation qui y est utilisable (i.e., la *complexité* des instances effectives de la RS correspondante) ; de même, les interrelations de garantie sont "qualifiées" par l'interrelation entre la *complexité* des transformations de programmes correspondant aux RS en cause.

Nous nous préoccupons d'aspects de complexité principalement concernant la puissance expressive de s-1-1 et l'interrelation de garantie de composition effective par s-1-1 (notre Chapitre 3). Sur le premier sujet, nous améliorons des résultats de Royer [Roy87], non seulement par le fait que nous touchons strictement plus de propriétés de programmation, mais également du strict point de vue complexité. Sur le deuxième sujet, nous invalidons l'intuition, exprimée par deux fois dans la littérature [MWY78, Roy87], à l'effet que cette interrelation de garantie est *nécessairement* inefficace du point de vue complexité. Nous donnons une construction *relativement efficace* de cette interrelation de garantie, et nous montrons que cette construction est optimale en général. Ces considérations de complexité nous amènent à identifier une classe spécifique de RVE comme étant au moins une "bonne approximation" de l'ensemble des techniques de programmation devant être *pratiquement* utilisables dans un système de programmation pour qu'on le dise "bon". Elles nous amènent aussi à conclure que la propriété de substitution effective est strictement plus fondamentale que celle de composition effective.

Les interrelations de garantie sont en général *non-constructives*, en ce sens que la preuve d'une telle interrelation ne permet pas de déduire qu'à partir d'un algorithme pour la transformation de programmes correspondant dans un système de programmation à la relation sémantique "garantissante", on peut *uniformément* trouver un algorithme pour la transformation correspondant, dans le même système de programmation, à la relation sémantique "garantie". Nous étudions quelque peu ce phénomène au Chapitre 5. Nous démontrons la non-constructivité d'une interrelation spécifique (s-1-1 garantit programmabilité), et nous montrons que la programmabilité garantit de façon constructive une vaste classe de RS. Ce dernier résultat suggère qu'un compilateur pour un langage standard constitue une information *nécessaire et suffisante* pour programmer dans un langage dont on connaît un interprète : nécessaire, parce que si l'on ne sait pas compiler les programmes d'un langage standard, alors une technique à la fois simple et fondamentale de programmation nous échappe ; suffisante parce que la connaissance d'un compilateur pour un langage standard nous permet d'implanter une vaste

classe d'autres techniques de programmation.

Un corollaire de la non-constructivité de l'interrelation "s-1-1 garantit programmabilité" est qu'on ne peut en général "inférer" (i.e., calculer) un compilateur pour un langage standard à partir d'un interprète pour un système de programmation acceptable (en fait, nous montrons que *même avec un oracle dans le problème d'arrêt*, on ne peut réaliser cette inférence). Ceci, couplé au fait qu'un compilateur pour un langage standard constitue une information nécessaire et suffisante pour programmer dans un langage, signifie intuitivement qu'on ne peut en général apprendre à programmer à partir du seul compilateur (ou interprète) d'un langage, et qu'il faut parfois faire appel à un "guide du programmeur". En d'autres mots, la *sémantique opérationnelle* (intuitivement, la description d'un interprète [Sto77, § 2]) n'est pas en général une description suffisante d'un langage pour un programmeur. La non-constructivité de "s-1-1 garantit programmabilité" nous dit en plus qu'un interprète *accompagné* d'une procédure uniforme pour substituer une constante à un paramètre formel dans un programme ne constitue pas *non plus* une description suffisante.

Nous étudions aussi au Chapitre 5 l'insolubilité de certains problèmes de décision sur les interprètes de SP.

1.2 Travaux connexes

La sémantique dénotationnelle [Sto77] de même que la théorie des schèmes de programmes [Gre75] donnent lieu à des notions de "rapport sémantique". Cependant, comme Royer fait remarquer [Roy87, § 1.1], ces notions ne correspondent qu'à ce que nous appelons les relations sémantiques extensionnelles sans récursion au sens de Royer. Elles n'ont donc pas à proposer de "grilles de référence" pour propriétés de programmation comme le formalisme de Riccardi ou celui des relations vérifiables par énumération. Notons que ces théories se veulent plutôt la base de disciplines de conception de langages, plutôt que des outils d'étude théorique des langages, et ne cherchent donc pas nécessairement à couvrir un grand nombre de techniques de programmation.

Nous incluons dans nos définitions les SP *sous-récursifs*, i.e., qui n'ont pas de programmes pour *toutes* les fonctions partielles récursives. Cependant, nous n'en faisons pas une étude systématique. Ces SP sont étudiés dans

[CB71, Alt80, Roy87, RC86, RoyCas], et quelque peu dans [MY78]. Un genre de résultat populaire sur les SP sous-récursifs est la *concision relative* des programmes [RC86, RoyCas].

C'est à notre connaissance dans [MWY78] qu'est utilisée pour la première fois l'expression "propriété de programmation".

Hartmanis et Baker [Har74, HB75] s'intéressent à la complexité des traductions entre SP, particulièrement des traductions bijectives, en relation avec la complexité de fonctions de rembourrage (*"padding"*). Ils introduisent entre autres dans [HB75] la classe des numérations de Gödel (systèmes de programmation) *polynomiales*, i.e., vers lesquelles n'importe quel autre SP peut être traduit en temps polynomial (classe dénommée $GN\mathcal{P}time$ par Young [You88]). Ils émettent la fameuse conjecture, dite de Hartmanis-Baker, selon laquelle tous ces SP sont isomorphes via des traductions bijectives calculables en temps polynomial, et dont les inverses sont aussi calculables en temps polynomial. Cette conjecture est toujours ouverte. Machtey, Winklmann et Young [MWY78] se sont aussi intéressés à la complexité des traductions.

La principale propriété de programmation qui n'est *pas* exprimable en sémantique dénotationnelle et qui est malgré tout beaucoup utilisée est sans contredit celle correspondant au *Théorème de récursion de Kleene* (reproduit ici comme Théorème 1.8). Cette propriété (notre Définition 2.27) est exprimable dans le formalisme de Riccardi et tous les SP acceptables en sont dotés. Intuitivement, une instance effective de la RS correspondant à cette propriété retourne des programmes auto-référenciels, où l'auto-référence n'est pas nécessairement *extensionnelle* (i.e., essentiellement limitée à un appel récursif), mais peut aussi être *intensionnelle*, correspondant conceptuellement au cas où le programme a accès à son propre *texte*. Cette propriété a une foule d'applications en calculabilité et en complexité (e.g., [Blu67, MY81, RoyCas]). Son étude est centrale dans [Ric80, Ric81, Ric82, Roy87].

Royer [Roy87] a développé des outils très puissants de preuves d'indépendance de structures de contrôle dans le contexte des SP exécutables maximaux. On trouve aussi dans [Roy87] un grand répertoire de caractérisations de l'acceptabilité.

Il est intéressant de noter que, bien que le formalisme de Riccardi et celui des RVE soient avant tout des outils d'étude théorique des langages de programmation, certaines RS exprimables dans ces formalismes et *pas* dans

le cadre de la sémantique dénotationnelle se rencontrent parfois dans des contextes "pratiques". Ainsi, on trouve dans [HNTJ89] la description d'un langage (réellement implanté) incorporant une construction primitive d'auto-référence *textuelle* (comme dans le Théorème de récursion de Kleene). Nous avons récemment appris que l'inversion fonctionnelle joue un rôle très important (avec de nombreuses applications) dans la théorie de la synthèse d'algorithmes à partir de spécifications fonctionnelles [Bir90]. Cela suggère la possibilité que la pseudo-inversion ait elle aussi des applications pratiques.

1.3 Préliminaires

1.3.1 Généralités

Le symbole $\stackrel{\mathrm{d}}{=}$ se lit "égale par définition".

La lettre N dénote l'ensemble des entiers non-négatifs (appelés simplement *entiers*), $\{0, 1, 2, \ldots\}$. Sauf indication contraire, toute lettre latine minuscule $(a, b, \ldots)$ *sauf* f, g et h, éventuellement décorée d'indices ou d'accents $(')$, dénote un entier. Une lettre latine majuscule $(A, B, \ldots)$, éventuellement décorée d'indices ou d'accents, dénote habituellement un sous-ensemble de N.

Pour tout $n > 0$, $\underline{n}$ dénote l'ensemble $\{x \mid 1 \leq x \leq n\}$ [Roy87].

Les symboles $\subset$ et $\supset$ dénotent des inclusions *strictes* d'ensembles, $\subseteq$ et $\supseteq$ dénotent des inclusions possiblement non-strictes. Nous dénotons la différence ensembliste par $-$. Si A est un ensemble quelconque, alors $\overline{A}$ dénote le *complément* de A, c'est-à-dire l'ensemble $U - A$, où U est l'univers de discours prévalant au moment de la définition de A. Si A dénote un sous-ensemble de N (comme c'est le cas à moins d'avis contraire), alors $\overline{A}$ dénote $N - A$. L'ensemble vide est dénoté par $\emptyset$. L'ensemble des *parties* (sous-ensembles) de A est dénoté par 2^A.

Si A et B sont des ensembles quelconques, alors $A \times B$ dénote le *produit cartésien* de A par B, c'est-à-dire l'ensemble des *couples* (ordonnés) (a, b) tels que $a \in A$ et $b \in B$. Plus généralement, soit $k > 0$, si $A_1, \ldots, A_k$ sont des ensembles quelconques, alors

$$A_1 \times \cdots \times A_k \stackrel{\mathrm{d}}{=} \{(a_1, \ldots, a_k) \mid a_1 \in A_1 \; \& \cdots \& \; a_k \in A_k\}.$$

On appelle les éléments de $A_1 \times \cdots \times A_k$ des *k-tuplets* (ordonnés), ou des tuplets *d'arité* k. Un couple est donc un 2-tuplet.

Pour tout $k > 1$, A^k dénote $A \times \cdots \times A$, où A apparaît k fois. Par convention, A^1 dénote A. On considère donc un "élément simple" comme un 1-tuplet.

La *longueur* d'un entier x, dénotée $|x|$, est le nombre de bits significatifs dans la représentation binaire de x. On note que pour tout x, $|x| = \lceil \lg(x+1) \rceil$. Le symbole lg dénote la fonction logarithme en base 2. Soient $k > 0$ et $(x_1, \ldots, x_k) \in N^k$, alors les expressions $|(x_1, \ldots, x_k)|$ et $|x_1, \ldots, x_k|$, dénotent toutes deux $|x_1| + \cdots + |x_k|$.

1.3.2 Fonctions, relations

Nous présentons maintenant les concepts de *relations* et de *fonctions partielles* sur N^k, pour un certain k. Nous suivons essentiellement [Rog87].

Soit $k > 0$. Tout sous-ensemble R de N^k est dit *relation sur* N^k, ou *k-aire*, ou *d'arité* k. Pour tout $x_1, \ldots, x_k$, on écrit $R(x_1, \ldots, x_k)$ ssi $(x_1, \ldots, x_k) \in R$. Une relation unaire est en fait un sous-ensemble de N et est parfois appelée un *prédicat* sur N. Si P est un prédicat sur N, alors l'expression $(\mu x)[P(x)]$ est indéfinie si $\neg(\exists x)[P(x)]$, et dénote autrement le *plus petit entier* x tel que $P(x)$.

Soient $k > 1$ et R une relation k-aire. On dit que R est *univaluée* ssi pour tout $x_1, \ldots, x_{k-1} \in N$, il existe *au plus un* $x_k \in N$ tel que $(x_1, \ldots, x_{k-1}, x_k) \in R$. Une relation k-aire univaluée est aussi dite *fonction partielle* $(k-1)$-aire, ou *d'arité* $k-1$, ou *sur* N^{k-1}, ou *de* N^{k-1} *dans* N.

Une *fonction partielle* est une fonction partielle k-aire pour un certain $k > 0$. Par défaut, les lettres grecques minuscules du début de l'alphabet (α, β, γ), éventuellement décorées d'indices ou d'accents, dénotent des fonctions partielles. Si l'arité d'une fonction partielle n'est pas implicitement ou explicitement donnée, elle est supposée être 1.

Soit $k > 0$ et soit α une fonction partielle k-aire. Le fait que α soit une fonction partielle k-aire s'écrit $\alpha : N^k \to N$. Pour tout $x_1, \ldots, x_k \in N$, on dit que α *est définie sur* (ou parfois *converge sur*) $(x_1, \ldots, x_k)$, situation dénotée par $\alpha((x_1, \ldots, x_k))\downarrow$, ou simplement $\alpha(x_1, \ldots, x_k)\downarrow$, ssi il existe un $x_{k+1} \in N$

tel que $(x_1, \ldots, x_k, x_{k+1}) \in \alpha$. Ssi $\neg[\alpha(x_1, \ldots, x_k)\downarrow]$, on dit que α est *indéfinie sur* (ou parfois *diverge sur*) $(x_1, \ldots, x_k)$, et on écrit $\alpha((x_1, \ldots, x_k))\uparrow$, ou simplement $\alpha(x_1, \ldots, x_k)\uparrow$. Le *domaine* de α, dénoté $\mathbf{dom}(\alpha)$, est l'ensemble $\{(x_1, \ldots, x_k) \in N^k \mid \alpha(x_1, \ldots, x_k)\downarrow\}$. Pour tout $(x_1, \ldots, x_k) \in \mathbf{dom}(\alpha)$, on dénote par $\alpha((x_1, \ldots, x_k))$, ou simplement $\alpha(x_1, \ldots, x_k)$, l'unique $x_{k+1} \in N$ tel que $(x_1, \ldots, x_k, x_{k+1}) \in \alpha$. On utilise parfois la notation $\alpha(x_1, \ldots, x_k)\downarrow$ pour à la fois signifier que $(x_1, \ldots, x_k) \in \mathbf{dom}(\alpha)$ *et* dénoter la valeur $\alpha(x_1, \ldots, x_k)$. Par exemple, $\alpha(x_1, \ldots, x_k)\downarrow \neq 2$ signifie que $\alpha(x_1, \ldots, x_k)\downarrow$, et que $\alpha(x_1, \ldots, x_k) \neq 2$.

Soient $k > 0$, α et β deux fonctions partielles k-aires, et $\vec{x} \in N^k$. La notation $\alpha(\vec{x}) = \beta(\vec{x})$ signifie que $\alpha(\vec{x})\downarrow \Longleftrightarrow \beta(\vec{x})\downarrow$ et que $\alpha(\vec{x})\downarrow \Longrightarrow \alpha(\vec{x})\downarrow = \beta(\vec{x})\downarrow$. La notation $\alpha(\vec{x}) \neq \beta(\vec{x})$ signifie que $\neg[\alpha(\vec{x}) = \beta(\vec{x})]$. On remarque que $\alpha = \beta$ ssi pour tout $\vec{x} \in N^k$, $\alpha(\vec{x}) = \beta(\vec{x})$. On dit que β est une *variante finie* de α ssi $\alpha(\vec{x}) = \beta(\vec{x})$ pour *presque tous les* $\vec{x} \in N^k$, c'est-à-dire pour tous les $\vec{x} \in N^k$, sauf un nombre fini d'entre eux.

Soient $k > 0$, $\alpha : N^k \to N$, et $A \subseteq \mathbf{dom}(\alpha)$. La notation $\alpha(A)$ dénote l'ensemble $\{\alpha(a) \mid a \in A\}$. L'*image* de α, dénotée $\mathbf{image}(\alpha)$, est l'ensemble $\alpha(\mathbf{dom}(\alpha))$. Ssi $\mathbf{dom}(\alpha) = N^k$, on dit que α est *totale*. Ssi $\mathbf{image}(\alpha) = N$, on dit que α est *surjective*. Ssi pour tout $y \in N$ il existe *au plus un* $\vec{x} \in \mathbf{dom}(\alpha)$ tel que $\alpha(\vec{x}) = y$, on dit que α est *injective*. On dit que α est *bijective* ssi elle est à la fois surjective et injective (attention : une fonction partielle bijective n'est pas nécessairement une *bijection*, notion définie au prochain paragraphe). Si α est une fonction partielle unaire injective, on dénote par α^{-1} l'unique fonction partielle β telle que $\mathbf{dom}(\beta) = \mathbf{image}(\alpha)$ et pour tout $x \in \mathbf{dom}(\alpha)$, $\beta(\alpha(x)) = x$.

Une fonction partielle *totale* est appelée une *fonction* (ou *application*). Par défaut, les lettres f, g et h dénotent des fonctions. Une fonction injective (respectivement, surjective, bijective) est appelée une *injection* (respectivement, *surjection*, *bijection*). Une *fonction finie* est une fonction partielle qui, comme ensemble de tuplets, est finie. Une fonction partielle est finie ssi son domaine est fini. Une *fonction constante* est une fonction (donc, totale) dont l'image est un singleton.

Nous utilisons souvent la notation *lambda* pour décrire des fonctions partielles [Rog87]. En résumé, si $k > 0$ et si "$\cdots x_1 \cdots x_k \cdots$" est une expression impliquant les variables $x_1, \ldots, x_k$, alors $\lambda x_1, \ldots, x_k.[\cdots x_1 \cdots x_k \cdots]$ dénote l'unique fonction partielle k-aire α telle que pour tout $x_1, \ldots, x_k \in N$,

$\alpha(x_1, \ldots, x_k) =$ "$\cdots x_1 \cdots x_k \cdots$". Intuitivement, "$\cdots x_1 \cdots x_k \cdots$" donne un algorithme pour calculer la fonction partielle décrite. Le symbole "$\uparrow$" utilisé dans une lambda-expression signifie "indéfini". Ainsi, par exemple, $\lambda z.\uparrow$ dénote la fonction partielle unaire indéfinie partout. Les crochets [ ] sont souvent omis.

Si α et β sont deux fonctions partielles (unaires), alors $\alpha \circ \beta$ dénote la *composition fonctionnelle* de α et β, i.e., la fonction partielle $\lambda x.\alpha(\beta(x))$. La composition fonctionnelle est associative. Par convention, α^0 dénote *l'identité* (la fonction $\lambda x.x$). Récursivement, pour tout $n > 0$, α^n dénote $\alpha \circ \alpha^{n-1}$. L'expression $\alpha \oplus \beta$ dénote la fonction partielle $\lambda x.[\alpha(x/2)$ si x est pair ; $\beta((x-1)/2)$ autrement]. L'opération $\oplus$ n'est pas associative, mais on sous-entend une association syntaxique à droite : pour toutes $\alpha_1, \ldots, \alpha_n$, $\alpha_1 \oplus \cdots \oplus \alpha_n$ dénote $\alpha_1 \oplus (\alpha_2 \oplus \cdots \oplus \alpha_n)$.

On dénote par $\mathcal{P}art$ l'ensemble des fonctions partielles de toutes arités ; $\mathcal{P}art\mathcal{U}n$ dénote l'ensemble des fonctions partielles (unaires) ; $\mathcal{T}ot$ dénote l'ensemble des fonctions (totales) de toutes arités.

Les concepts associés aux relations et fonctions partielles sur N^k pour un certain k, y compris la notation lambda, se généralisent facilement et naturellement à des ensembles quelconques de tuplets, et nous utilisons parfois ces généralisations.

1.3.3 Notations asymptotiques

Soient $n > 0$ et f une fonction n-aire. On définit

$$O(f) \stackrel{\mathrm{d}}{=} \{g : N^n \to N \mid (\exists c)(\forall \vec{x} \in N^n)[g(\vec{x}) \leq c \cdot (f(\vec{x}) + 1)]\}$$

et

$$\begin{aligned}\Omega(f) \stackrel{\mathrm{d}}{=} \{g : N^n \to N \mid (\exists c)(\forall x_0)(\exists x_1, \ldots, x_n)[(x_1 \geq x_0 \mathbin{\&} \cdots \mathbin{\&} x_n \geq x_0) \\ \mathbin{\&} c \cdot (g(x_1, \ldots, x_n) + 1) \geq f(x_1, \ldots, x_n)]\}.\end{aligned}$$

Nous attirons l'attention du lecteur sur le fait que ces définitions diffèrent d'autres définitions retrouvées dans la littérature, par exemple dans [BB88]. Nos définitions sont simplement mieux adaptées aux usages spécifiques que nous en faisons. Nous ne prétendons à aucune symétrie ou complémentarité entre nos définitions de O et Ω.

1.3.4 Modèle de calcul, complexité

Le modèle de calcul que nous utilisons est la *machine de Turing déterministe* [HU79]. Si nous ne faisions que de la calculabilité, nous en aurions dit suffisamment ; cependant, nous ferons aussi de la complexité, et il nous faut donc préciser quelques détails. Nous serons beaucoup préoccupés, au Chapitre 3, de calculs sur plusieurs entrées faisables en temps linéaire. Le but des détails donnés ici est de nous assurer que ce "temps linéaire" est bien défini. Nous supposons le lecteur familier avec la terminologie des machines de Turing et avec au moins un de leurs modes de fonctionnement standard [HU79].

Nos machines sont multi-ruban. Leur alphabet d'entrée et de sortie est $\{0, 1\}$. Chaque ruban est infini vers la droite (le lecteur verra à se choisir une droite). Dans notre discussion des machines, nous considérons que les rubans d'une machine sont numérotés consécutivement à partir de 1, selon un ordre déterminé sans équivoque. Une machine a toujours au moins 2 rubans. Chaque machine a une *arité* fixe et unique, strictement positive et inférieure à son nombre de ruban. L'arité d'une machine donne à la fois le nombre d'entrées qu'elle accepte simultanément pour les fins d'un seul et même calcul, et (conséquemment) l'arité de la fonction partielle qu'elle calcule.

Soit M une machine à k rubans et d'arité n. On note que n satisfait forcément $1 \leq n < k$, et que M calcule une fonction partielle n-aire, i.e., de N^n dans N. Pour "utiliser" M, les n entrées sont fournies sur les rubans 1 à n respectivement, et la sortie est récupérée du ruban k si et lorsque M s'arrête *normalement* (voir plus bas), et est considérée absente autrement. On appelle les rubans 1 à n les *rubans d'entrée*, et le ruban k le *ruban de sortie*. Les autres rubans (s'il y en a) sont dits rubans *de travail*.

La tête du ruban de sortie est une tête d'écriture seulement, et ne peut écrire que des 1 et des 0 (en particulier, pas de blanc). À toute étape de calcul, elle ne peut que se déplacer vers la droite ou rester stationnaire. Les étapes pendant lesquelles elle écrit un symbole sur le ruban sont exactement celles à la fin desquelles elle se déplace. Les têtes des rubans d'entrée sont des têtes de lecture seulement, mais elles peuvent se déplacer (dans les deux sens) ou rester stationnaires à toute étape de calcul. Les têtes des rubans de travail n'ont aucune restriction spéciale ; en particulier, elles peuvent se déplacer ou rester stationnaires à toute étape de calcul, qu'elles aient ou non écrit un symbole au cours de cette étape.

Les entrées sont des entiers donnés en représentation binaire *réduite* (i.e., sans bits non-significatifs) inversée (i.e., le bit le *moins* significatif à gauche), flanquées à gauche et à droite d'un symbole marqueur spécial, dans les cases les plus à gauche de leur ruban respectif, avec la tête reposant initialement sur la case immédiatement à droite du marqueur de gauche, i.e., sur le bit le moins significatif de l'entrée, ou sur le marqueur de droite, si la valeur de l'entrée est 0. Notons que le symbole marqueur spécial est obligatoirement élément de l'alphabet de ruban d'une machine, mais il n'est pas considéré comme faisant partie de son alphabet d'entrée, qui est toujours $\{0, 1\}$. Une tête de ruban d'entrée ne doit jamais se déplacer à droite d'un marqueur de droite. (On peut dire, par exemple, que la machine s'arrête dès qu'une tête de ruban d'entrée repose sur une case à blanc.) La sortie doit être donnée en représentation binaire réduite inversée, sans marqueurs bien sûr, puisque seuls les symboles 0 et 1 peuvent être écrits sur le ruban de sortie.

On dit qu'une machine s'arrête *normalement* ssi un entier en représentation binaire réduite inversée se trouve sur son ruban de sortie au moment où elle s'arrête (i.e., s'il n'y a *pas* un 0 immédiatement à gauche de la tête du ruban de sortie au moment de l'arrêt). Tel que dit précédemment, l'arrêt anormal d'une machine est interprété comme le défaut de s'arrêter, et ne produit *pas* de sortie.

Sujet aux conventions exposées ci-dessus, le fonctionnement de nos machines suit les règles habituelles de fonctionnement des machines de Turing. La correspondance entre une machine et la fonction partielle qu'elle calcule se fait de la façon naturelle, avec l'absence de sortie correspondant au fait que la fonction calculée n'est pas définie pour la combinaison d'entrées soumise.

L'expression "l'entrée soumise (au singulier) à une machine" réfère au tuplet composé des entrées soumises à une machine pour les fins d'un calcul. Sauf avis contraire, donc, la "longueur de l'entrée soumise" à une machine correspond à la longueur du tuplet d'entrées soumis. On note que la longueur d'une entrée consistant en un tuplet $(x_1, \ldots, x_n)$ est dénotée par $|(x_1, \ldots, x_k)|$ ou encore $|x_1, \ldots, x_n|$, et correspond à la somme des longueurs des entrées telles que présentées à la machine, excluant les marqueurs.

Le *temps* de calcul d'une machine sur une entrée donnée est considéré infini si la machine ne fournit pas de sortie, et est autrement le nombre d'étapes de calcul complétées entre le départ et l'arrêt de la machine. (Notons que les fonctions constantes $\lambda x_1, \ldots, x_n.0$, pour tout $n > 0$, sont calculables en

temps 0 peu importe l'entrée, un ruban de sortie "vide" représentant l'entier 0).

Suivant [HU79], *l'espace* de calcul d'une machine sur une entrée donnée est considéré infini si la machine ne fournit pas de sortie, et est autrement le maximum, pris sur tous les rubans de travail de la machine (donc, excluant les rubans d'entrée et de sortie), du nombre de cases visitées au cours du calcul. S'il n'y a pas de ruban de travail, alors l'espace est considéré nul.

Le temps (l'espace) de calcul d'une machine est habituellement donné globalement pour toutes les entrées possibles sous forme d'une fonction partielle T de même arité que la machine et telle que pour tout $\vec{x} \in N^k$, où k est l'arité de la machine, $[T(\vec{x})\uparrow \iff$ le temps (l'espace) de calcul sur entrée $\vec{x}$ est infini], et $[T(\vec{x})\downarrow \implies$ le temps (l'espace) de calcul sur entrée $\vec{x}$ est $T(\vec{x})$].

Pour exprimer des temps (espaces) de calcul, on utilise souvent la variable n pour dénoter la "longueur de l'entrée", sans égard à l'arité de la machine. Ce symbole est donc *surchargé* (*"overloaded"*) dans l'expression de temps (espaces) de calcul, et sa signification est déterminée par la machine ou la fonction partielle à laquelle s'applique le temps (l'espace) de calcul exprimé. Ainsi, par exemple, une machine d'arité k a un temps de calcul dans $O(n)$ ssi son temps de calcul est dans $O(\lambda x_1, \ldots, x_k.|x_1, \ldots, x_k|)$.

Comme d'habitude, un temps (espace) de calcul est dit *au plus linéaire* (respectivement, *au plus polynomial*) ssi il est dans $O(n)$ (respectivement, $O(p(n))$ pour un certain polynôme p). Notons que, suivant [Roy87], un temps (espace) de calcul est dit *au plus exponentiel* ssi il est dans $O(2^{p(n)})$ pour un certain polynôme p.

Soit "$\cdots$" un expression pouvant se rapporter à un temps (espace) de calcul. Une fonction partielle α est dite *calculable en temps (espace)* "$\cdots$" ssi il existe une machine de Turing qui calcule α et dont le temps (l'espace) de calcul est au plus "$\cdots$". Un ensemble est *décidable en temps (espace)* "$\cdots$" ssi sa *fonction caractéristique* (§ 1.3.5) est calculable en temps (espace) "$\cdots$".

Nous supposons le lecteur familier avec les techniques standard de construction de machines de Turing, comme la queue de colombe (*"dovetail"*), le marquage de certaines cases d'un ruban de travail et la détection de la première visite d'une case d'un ruban de travail. Nous utilisons ces techniques sans introduction. Le lecteur est référé à [MY78, HU79] pour plus d'explications.

Au lieu de donner des descriptions précises de machines de Turing, nous donnons des algorithmes, parfois en français, mais le plus souvent dans un pseudo-langage dont la lecture ne devrait poser aucun problème à quiconque est familier avec un langage impératif courant (e.g., PASCAL [LN85]). Précisons malgré tout que le symbole "$\leftarrow$" dénote l'assignation d'une variable, et que l'instruction "retourner [*expression*]" signifie de procéder à l'évaluation de "[*expression*]" et, advenant le cas où l'évaluation se termine et donne un résultat, produire ce résultat comme sortie et s'arrêter immédiatement. Autrement, l'exécution de l'instruction ne se termine pas.

Lorsqu'un algorithme doit démontrer la complexité d'une fonction ou d'un ensemble, nous l'exposons en termes plus près des machines de Turing que s'il doit simplement en démontrer la récursivité ou la décidabilité. Un algorithme donné dans le cadre d'une lambda-définition de fonction est souvent considéré tacitement comme démontrant la récursivité (du moins partielle) de la fonction définie.

On dénote par $\mathcal{L}intime$ l'ensemble des fonctions de toutes arités calculables en temps linéaire ; $\mathcal{P}time$ dénote l'ensemble des fonctions de toutes arités calculables en temps polynomial ; $\mathcal{E}xptime$ dénote l'ensemble des fonctions de toutes arités calculables en temps exponentiel ; $\mathcal{P}rim\mathcal{R}ec$ dénote l'ensemble des fonctions *primitives récursives* [MY78, Rog87] de toutes arités ; $\mathcal{E}lm\mathcal{R}ec$ dénote l'ensemble des fonctions *élémentaires récursives* [BL74, MY78] de toutes arités. Très brièvement, une fonction est élémentaire récursive ssi elle possède une dérivation primitive récursive dans laquelle n'interviennent que des fonctions bornées par $ex(n,k)+k$ pour un certain k fixe, où $ex(n,0) \stackrel{\mathrm{d}}{=} n$ et $ex(n,k+1) \stackrel{\mathrm{d}}{=} 2^{ex(n,k)}$.

1.3.5 Calculabilité I

On dénote par $\mathcal{P}art\mathcal{R}ec$ l'ensemble des fonctions *partielles récursives* [Rog87] (intuitivement, les fonctions partielles "calculables") de toutes arités ; $\mathcal{P}art\mathcal{R}ec\mathcal{U}n$ dénote l'ensemble des fonctions partielles récursives (unaires).

La *fonction caractéristique* d'un ensemble A (sous-ensemble de N), dénotée c_A, est la fonction $\lambda z.[1 \text{ si } z \in A\,;\ 0 \text{ autrement}]$.

Un ensemble est dit *récursivement énumérable* (abrégé *r.é.*) ssi il est vide ou bien l'image d'une fonction récursive. Un ensemble est dit *récursif* ssi

sa fonction caractéristique est récursive. Il est bien connu qu'un ensemble est r.é. ssi il est le domaine d'une fonction partielle récursive. Un ensemble est *immunisé* [Rog87] ssi il est infini et ne possède aucun sous-ensemble r.é. infini. Un ensemble est *simple* ssi il est r.é. et son complément est immunisé. Un ensemble A est *hyperimmunisé* ssi $A = \{a_0 < a_1 < \cdots\}$ est infini et pour toute f récursive, $f(n) < a_n$ infiniment souvent.

Une *fonction de pairage* est une bijection récursive de N^2 dans N. Dans tout cet ouvrage, nous utilisons la fonction de pairage de [RC86, Roy87], dénotée $\langle \cdot, \cdot \rangle$, et définie pour tout x, y par *l'entrelacement* des bits de x et y, i.e., plus précisément par

$$\langle x, y \rangle = \sum_{i \in X} 2^{2i+1} + \sum_{j \in Y} 2^{2j},$$

où X est l'ensemble des positions des 1 dans la représentation binaire de x, et Y est la même chose pour y. Par exemple, si $x = 15$ et $y = 2$, alors $X = \{0, 1, 2, 3\}$, $Y = \{1\}$ et $\langle 15, 2 \rangle = \sum_{k \in \{1,3,5,7,2\}} 2^k = 174$.

Récursivement, pour tout $k > 2$, on définit $\langle x_1, \ldots, x_k \rangle$ comme étant $\langle x_1, \langle x_2, \ldots, x_k \rangle \rangle$. Par convention, $\langle x \rangle$ dénote x pour tout $x \in N$.

On appelle souvent "paire" un entier résultant d'une ou plusieurs applications de la fonction de pairage, pour souligner que, conceptuellement, cet entier en encode plus d'un.

On élargit la notation lambda comme suit : soit $k > 1$, et soit "$\cdots x_1 \cdots x_k \cdots$" une expression impliquant les variables $x_1, \ldots, x_k$. Alors $\lambda \langle x_1, \ldots, x_k \rangle.[\cdots x_1 \cdots x_k \cdots]$ dénote la fonction partielle $\lambda z.[\cdots x_1 \cdots x_k \cdots$, où $\langle x_1, \ldots, x_k \rangle = z]$, où z est une variable n'apparaissant pas dans "$\cdots x_1 \cdots x_k \cdots$".

La *première fonction de projection* de $\langle \cdot, \cdot \rangle$, que nous dénotons habituellement π_1, est la fonction $\lambda \langle x, y \rangle.x$. La *seconde fonction de projection* de $\langle \cdot, \cdot \rangle$, dénotée habituellement π_2, est la fonction $\lambda \langle x, y \rangle.y$. Les trois fonctions $\langle \cdot, \cdot \rangle$, π_1 et π_2 sont calculables en temps linéaire [RC86]. On note que la composition d'un nombre fixe de ces fonctions est aussi calculable en temps linéaire. Par exemple, c'est le cas de $\lambda x, y, z.\langle x, y, z \rangle$.

On montre facilement que :

1.1 Proposition (a) $\langle \cdot, \cdot \rangle$ *est strictement croissante en chaque argument.*

(b) *Pour tout* a *et* b, $2 \cdot \max(|a|, |b|) - 1 \leq |\langle a, b\rangle| \leq 2 \cdot \max(|a|, |b|)$, *avec égalité à gauche si* $|a| < |b|$ *et à droite autrement.*

(c) *Pour tout* $a > 1$ *et tout* b, $\langle a, b\rangle > \max(a, b)$.

Il pourra nous arriver d'utiliser la Proposition 1.1 sans y référer explicitement.

Si α est une fonction partielle k-aire pour un $k > 1$, alors la *version unarisée* de α est la fonction partielle unaire $\lambda\langle x_1, \ldots, x_k\rangle.\alpha(x_1, \ldots, x_k)$.

Soit $n > 0$, et soit $\tau \stackrel{\mathrm{d}}{=} \lambda x_1, \ldots, x_n.\langle x_1, \ldots, x_n\rangle$. Une relation n-aire R est dite récursive ssi l'ensemble $\tau(R)$ est récursif. Pour tout $k > 0$, on dit que R est *exprimable par une forme* Σ_k^0 (respectivement, Π_k^0) ssi il existe un entier $m > 0$, une relation récursive $(n+m)$-aire S et une séquence $(Q_i y_i)$, $i \in \underline{m}$ de quantifications avec $k - 1$ *alternances*, tels que Q_1 est le quantificateur $\exists$ (respectivement, $\forall$) et pour tout $x_1, \ldots, x_n$, $R(x_1, \ldots, x_n) \iff (Q_1 y_1) \ldots (Q_m y_m)[S(x_1, \ldots, x_n, y_1, \ldots, y_m)]$. Pour tout $k > 0$, Σ_k^0 (respectivement, Π_k^0) dénote l'ensemble des relations (de toutes arités) exprimables par une forme Σ_k^0 (respectivement, Π_k^0). De plus, Σ_0^0 et Π_0^0 dénotent l'ensemble des relations récursives de toutes arités. Les relations unaires de Σ_1^0 correspondent aux ensembles r.é. [Rog87, § 14.1].

1.3.6 Systèmes de programmation

1.2 Définition Un *système de programmation* est une application (totale) de N dans $\mathcal{P}art\mathcal{R}ec\mathcal{U}n$.

On abrège *système de programmation* par *SP*. Les lettres grecques ϕ, ρ et ψ, éventuellement décorées d'accents ou d'exposants, dénotent des SP ; ϕ dénote un SP spécifique (voir ci-dessous). L'appellation "système de programmation" est due à Machtey et Young [MY78]. On rencontre souvent aussi les appellations *"numbering"* ou *"indexing of the partial recursive functions"*.

Soit ψ un SP. Tout entier p est un *programme en* ψ, ou un ψ*-programme*. Pour tout p, on dénote souvent $\psi(p)$ par ψ_p. Cette notation nous permet d'écrire des expressions du genre $(\psi(p))(x)$ comme $\psi_p(x)$. On dit que p *calcule* ψ_p, ou *est un programme pour* ψ_p. La *fonction universelle* de ψ est la fonction partielle $\lambda p, x.\psi_p(x)$. Sa fonction universelle *unaire*, dénotée $\widehat{\psi}$,

est la fonction partielle unaire $\lambda\langle p, x\rangle.\psi_p(x)$. La notion de SP dont on peut interpréter uniformément les programmes est définie comme suit.

1.3 Définition (Rogers [Rog58]) Un SP est dit *exécutable* ssi sa fonction universelle est partielle récursive.

Le terme “exécutable” est dû à Riccardi [Ric80]. Royer [Roy87] utilise “effectif”, Rogers [Rog58] utilise “semi-effectif”.

L'image d'un SP correspond intuitivement à sa *puissance de calcul*. Un SP surjectif (i.e., dont l'image est $\mathcal{P}art\mathcal{R}ec\mathcal{U}n$) est dit *de puissance de calcul maximale*, ou simplement *maximal*.

Nous définissons maintenant ϕ, un SP qui nous servira de SP “standard” tout au long de ce travail. Supposons fixé un schème de codage *bijectif* de toutes les machines de Turing en entiers (voir par exemple [HU79]), représenté par une bijection κ de l'ensemble des machines de Turing dans N. On suppose κ “raisonnable”, en ce sens qu'on puisse, à partir de n'importe quel code, retrouver en temps linéaire les principales caractéristiques de la machine correspondante, par exemple son arité. Pour tout i, M_i dénote la machine de Turing $\kappa^{-1}(i)$. On définit alors ϕ comme suit.

1.4 Définition Pour tout p, $\phi(p) \stackrel{\mathrm{d}}{=}$ la version unarisée de la fonction partielle calculée par M_p.

On vérifie aisément que ϕ est maximal. Il est bien connu que $\widehat{\phi}$ (la fonction universelle unaire de ϕ) est partielle récursive, et que donc, ϕ est exécutable.

Nous définissons maintenant quelques *propriétés de programmation* des SP. Ces propriétés seront définies plus tard en termes de *relations sémantiques* (Chapitre 2), et nous utilisons dès maintenant cette terminologie afin d'éviter une redondance inutile dans la nomenclature. Il n'est ni nécessaire ni possible pour le lecteur, à ce stade-ci, d'interpréter les expressions “instance effective de ...” autrement que comme des noms un peu sophistiqués donnés à certains objets formels. Cependant, nous verrons au Chapitre 2 que ces expressions, lorsque interprétées dans le contexte des *relations sémantiques*, ont bel et bien la signification qui est présentée ici.

1.5 Définition (Riccardi [Ric80]) Un SP ψ est dit *programmable* ssi il existe une fonction récursive t telle que pour tout i, $\psi_{t(i)} = \phi_i$. Pour toute telle t, on dit que ψ est programmable *via* t, et on dit que t est une *instance effective de* prog *en* ψ, ou une *fonction de programmation pour* ψ.

Notons qu'un SP programmable est nécessairement maximal, et que ϕ est banalement programmable, entre autres via la fonction identité.

1.6 Définition Un SP est dit *acceptable* ssi il est exécutable et programmable.

Le terme "acceptable" est probablement dû à Rogers [Rog67]. Le SP ϕ est bien sûr acceptable. Nous mentionnons quelques propriétés des SP acceptables.

Le *Théorème d'isomorphisme de Rogers* s'énonce comme suit.

1.7 Théorème (Rogers [Rog58]) *Soient ψ et ρ deux SP acceptables. Alors, il existe une bijection récursive t telle que pour tout p, $\psi_{t(p)} = \rho_p$.*

Une *instance effective de* krt *dans* un SP ψ est une fonction récursive f telle que pour tout p,

$$\psi_{f(p)} = \lambda x.[\psi_p(\langle f(p), x\rangle)].$$

Le *Théorème de récursion de Kleene* [Rog87, p. 214] s'énonce comme suit.

1.8 Théorème (Kleene [Kle38]) *Tout SP acceptable possède une instance effective de* krt.

Notons qu'une instance effective de krt retourne des programmes *auto-référenciels*, en ce qu'ils ont (informellement) accès à leur propre texte. On se permet souvent, en invoquant le Théorème de récursion de Kleene, de donner une description auto-référencielle d'un programme.

Pour tout $m > 0$ et $n > 0$, une *instance effective de* s-m-n *dans* un SP ψ est une fonction récursive $(m+1)$-aire f telle que pour tout p et $x_1, \ldots, x_m$,

$$\psi_{f(p,x_1,\ldots,x_m)} = \lambda\langle y_1, \ldots, y_n\rangle.[\psi_p(\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle)].$$

Le *Théorème s-m-n* [Rog87, Théorème 1.V] se lit comme suit.

1.9 Théorème (Kleene) *Soit ψ un SP acceptable. Alors, pour tout $m > 0$ et $n > 0$, ψ possède une instance effective de* s-m-n.

Observons que par notre définition de $\langle \cdot, \cdots, \cdot \rangle$ (§ 1.3.5), une instance effective de s-m-n dans un SP ψ est aussi une instance effective de s-m-n' dans ψ pour tout $n' > 0$. Pour cette raison, nous ne parlerons de s-m-n que pour $n = 1$.

Une *instance effective de* remb-inf dans un SP ψ est une fonction récursive 2-aire f telle que (i) pour tout p, $\lambda n.f(p,n)$ est injective, et (ii) pour tout p et tout n,

$$\psi_{f(p,n)} = \psi_p \;\&\; f(p,n) \neq p.$$

Intuitivement, une instance effective de remb-inf permet de grossir à volonté les programmes sans en changer le sens. Le *Lemme du rembourrage* (légère variante de [MY78, Proposition 3.4.5]) nous dit que :

1.10 Théorème *Tout SP acceptable possède une instance effective de* remb-inf.

Le fait qu'un SP ait une instance effective de s-1-1 (respectivement, prog) correspond au fait qu'il soit doté de la propriété s-1-1 (respectivement, de la propriété de programmabilité) mentionnée en § 1.1.

Nous définissons maintenant certaines classes de SP.

1.11 Définition (a) $\mathcal{SP} \stackrel{\mathrm{d}}{=} \{\psi \mid \psi \text{ est un SP}\}$.

(b) $\mathcal{SPE} \stackrel{\mathrm{d}}{=} \{\psi \mid \psi \text{ est un SP exécutable}\}$.

(c) $\mathcal{SPM} \stackrel{\mathrm{d}}{=} \{\psi \mid \psi \text{ est un SP maximal}\}$.

(d) $\mathcal{SPEM} \stackrel{\mathrm{d}}{=} \{\psi \mid \psi \text{ est un SP exécutable maximal}\}$.

(e) $\mathcal{SPP} \stackrel{\mathrm{d}}{=} \{\psi \mid \psi \text{ est un SP programmable}\}$.

(f) $\mathcal{SPA} \stackrel{\mathrm{d}}{=} \{\psi \mid \psi \text{ est un SP acceptable}\}$.

Il est bien connu que $\mathcal{SPA} \subset \mathcal{SPP} \subset \mathcal{SPM} \subset \mathcal{SP}$, que $\mathcal{SPA} \subset \mathcal{SPEM} \subset \mathcal{SPE} \subset \mathcal{SP}$, que $\mathcal{SPEM} \subset \mathcal{SPM}$, que $\mathcal{SPEM} \cup \mathcal{SPP} \neq \mathcal{SPM}$ et que $\mathcal{SPE} \cup \mathcal{SPM} \neq \mathcal{SP}$. Également, $\mathcal{SPA} = \mathcal{SPE} \cap \mathcal{SPP}$, par définition d'acceptabilité (Définition 2.31). Pour référence future, nous démontrons que $\mathcal{SPEM} - \mathcal{SPA} \neq \emptyset$.

1.12 Théorème *Il existe un SP exécutable maximal non-acceptable.*

Preuve. Soit ψ défini par

$$\psi(\langle a, b\rangle) \stackrel{\mathrm{d}}{=} \begin{cases} \lambda x.\uparrow & \text{si } \phi_a(b)\uparrow, \\ \phi_a & \text{autrement.} \end{cases}$$

Informellement, pour interpréter un ψ-programme $\langle a, b\rangle$ sur une entrée x, on effectue le calcul $\phi_a(b)$ et, si et quand un résultat est obtenu, on retourne la valeur $\phi_a(x)$. Donc, ψ est exécutable. Pour toute $\alpha \in \mathcal{PartRecUn}$, il existe un i tel que $\phi_i = \alpha$. Si α est indéfinie partout, alors, $\psi(\langle i, 0\rangle) = \alpha$, sinon, il existe un b tel que $\alpha(b)\downarrow$, et alors, $\psi(\langle i, b\rangle) = \alpha$. Dans les deux cas, il existe un ψ-programme calculant α. Donc, ψ a puissance de calcul maximale.

Nous montrons maintenant que ψ n'est pas programmable. Supposons le contraire et soit f une fonction de programmation pour ψ. Par le Théorème de récursion de Kleene (Théorème 1.8), il existe un ϕ-programme e réalisant l'algorithme suivant, où π_2 est la seconde fonction de projection de $\langle \cdot, \cdot \rangle$:

> **Entrée :** x
> **Algorithme pour ϕ_e :**
> Si $x = \pi_2(f(e))$, alors $\uparrow$, autrement retourner 0.
>
> □ **Algorithme**

Clairement, ϕ_e est la fonction constante 0 "trouée" au point $x_0 = \pi_2(f(e))$. On a donc $\phi_e(x_0)\uparrow$. Puisque f est une fonction de programmation pour ψ, $\psi_{f(e)} = \phi_e$. Soit $\langle p_0, x_0\rangle = f(e)$. Comme $\psi_{\langle p_0, x_0\rangle} = \phi_e \neq \lambda z.\uparrow$, alors clairement par la définition de ψ, $\phi_{p_0}(x_0)\downarrow$ et $\psi_{\langle p_0, x_0\rangle} = \phi_{p_0}$. On a donc $\phi_{p_0}(x_0)\downarrow = \psi_{\langle p_0, x_0\rangle}(x_0)\downarrow = \psi_{f(e)}(x_0)\downarrow = \phi_e(x_0)\downarrow$, ce qui contredit le fait que $\phi_e(x_0)\uparrow$. □

Le SP de la preuve précédente est dû à Hervé Caussinus. Nous aurons aussi besoin des deux faits suivants.

1.13 Proposition *Toute variante finie du SP ψ de la preuve du Théorème 1.12 est exécutable, maximal et non-acceptable.*

Preuve. Soient ψ comme dans l'énoncé du théorème et ρ une variante finie de ψ. Clairement, ρ est exécutable et, puisque chaque fonction partielle récursive est calculée par une infinité de ϕ- (et donc, de ψ-) programmes, ρ est maximal. Supposons qu'il soit programmable via f, une fonction récursive. Alors, ρ est acceptable. Par le Théorème 1.10, il existe r, une instance effective de remb-inf dans ρ. Soit n_0 tel que pour tout $n > n_0$, $\rho_n = \psi_n$. Définissons

$$g \stackrel{\text{d}}{=} \lambda p.[r(f(p), (\mu x)[r(f(p), x) > n_0])].$$

On vérifie aisément que g est aussi une fonction de programmation pour ρ. Or, comme tous les ρ-programmes dans $\mathbf{image}(g)$ excèdent n_0, ils ont la même sémantique en ψ et en ρ. On conclut que g est aussi une fonction de programmation pour ψ, et donc que ψ est acceptable : contradiction du Théorème 1.12. On déduit que ρ n'est pas programmable, et donc que ρ est non-acceptable. □

1.14 Proposition *Toute variante finie d'un SP acceptable est acceptable.*

Preuve. Soient ψ un SP acceptable, ρ une variante finie de ψ, et n_0 tel que pour tout $n > n_0$, $\psi_n = \rho_n$. Clairement, ρ est exécutable. Soient f et r respectivement une fonction de programmation et une instance effective de remb-inf pour ψ. Soit g telle que définie dans la preuve de la Proposition 1.13. On vérifie aisément que g est une fonction de programmation pour ρ, et on conclut que ρ est programmable et, partant, acceptable. □

1.3.7 Calculabilité II, opérateurs récursifs

Pour tout i, $W_i \stackrel{\text{d}}{=} \mathbf{dom}(\phi_i)$. On définit $K \stackrel{\text{d}}{=} \{i \mid i \in W_i\}$. L'ensemble K est une incarnation formelle du "problème d'arrêt". Il est bien connu que K est r.é. et non-récursif ($K \in \Sigma^0_1 - \Sigma^0_0$) [Rog87]. Par analogie avec le SP ϕ, on définit pour chaque i, ϕ^K_i comme étant la fonction partielle calculée par la machine de Turing M_i *avec oracle dans* K, et $W^K_i \stackrel{\text{d}}{=} \mathbf{dom}(\phi^K_i)$.

Pour chaque u, D_u dénote l'ensemble des positions des 1 dans la représentation binaire de u. Plus précisément, D_u est l'unique ensemble (fini) A tel

que $\sum_{i\in A} 2^i = u$. On dit que u est *l'indice canonique* de D_u. Il y a une correspondance bijective entre les ensembles finis et les indices canoniques.

Soit $\tau \overset{\text{d}}{=} \langle\cdot,\cdot\rangle = \lambda x, y.\langle x, y\rangle$, notre fonction de pairage. On note que pour toute $\alpha \in \mathcal{P}art\mathcal{U}n$, $\tau(\alpha) \subseteq N$. En effet, α est un ensemble de couples, et donc $\tau(\alpha)$ dénote $\{\tau(x,y) \mid (x,y) \in \alpha\}$.

Pour chaque i, *l'opérateur d'énumération déterminé par* W_i, dénoté Φ_i, est l'application (unique) $\Phi : 2^N \to 2^N$ telle que pour tout $B \subseteq N$,

$$\Phi(B) = \{x \mid (\exists u)[D_u \subseteq B \;\&\; \langle u, x\rangle \in W_i]\}.$$

Intuitivement, un élément $\langle u, x\rangle$ de W_i représente la règle de transformation d'ensembles : "dès que D_u est inclus dans l'ensemble de départ, on met x dans l'ensemble image". Pour cette raison, nous référons à chaque membre $\langle u, x\rangle$ de W_i, comme à la *règle* "D_u cause x". Si D_u est un singleton $\{s\}$, on appelle aussi $\langle u, x\rangle$ la règle "s cause x". Si $D_u = \tau(\alpha)$ pour une certaine fonction finie α on appelle aussi $\langle u, x\rangle$ la règle "α cause x".

Un *opérateur récursif* est une application $\Theta : \mathcal{P}art\mathcal{U}n \to \mathcal{P}art\mathcal{U}n$ telle qu'il existe un opérateur d'énumération Φ_i satisfaisant

$$(\forall\alpha)[\Theta(\alpha) = \tau^{-1}(\Phi_i(\tau(\alpha)))].$$

On dit que Θ est *déterminé par* Φ_i. Notons que les fonctions τ et τ^{-1} utilisées conjointement avec un opérateur d'énumération ne servent en fait qu'à "convertir" des couples en paires et vice-versa. Nous introduirons d'ailleurs sous peu des conventions de notation qui permettront de rendre implicite leur utilisation (Notation 1.16).

Les symboles Θ et Γ, éventuellement décorés d'indices ou d'accents, dénotent des opérateurs récursifs ; Φ et Ψ, éventuellement décorés d'indices ou d'accents, dénotent le plus souvent des opérateurs d'énumération et à l'occasion des opérateurs récursifs.

Nous utiliserons beaucoup les propriétés suivantes des opérateurs d'énumération (et donc des opérateurs récursifs).

1.15 Théorème [Rog87, Théorème 9.XXI] *Soit Φ un opérateur d'énumération.*

(a) *(Monotonicité) : Pour tout A et B, si $A \subseteq B$, alors $\Phi(A) \subseteq \Phi(B)$.*

(b) *(Continuité) : Pour tout x et tout A, si $x \in \Phi(A)$, alors il existe un D fini tel que $D \subseteq A$ et $x \in \Phi(D)$.*

Nous utilisons certaines conventions de notation pour les opérateurs d'énumération et les opérateurs récursifs. La plupart viennent de [Ric80, Roy87].

1.16 Notation Soient Θ un opérateur récursif et Φ un opérateur d'énumération. Soit aussi $\tau \stackrel{\mathrm{d}}{=} \langle \cdot, \cdot \rangle$.

(a) Pour tout $\psi \in \mathcal{SP}$,

$$\Theta(\psi) \stackrel{\mathrm{d}}{=} \Theta(\widehat{\psi}) \quad \text{et} \quad \Phi(\psi) \stackrel{\mathrm{d}}{=} \tau^{-1}(\Phi(\tau(\widehat{\psi}))).$$

(b) Pour tout $n > 0$ et toutes $\alpha_1, \ldots, \alpha_n \in \mathcal{P}art\mathcal{U}n$,

$$\Theta(\alpha_1, \ldots, \alpha_n) \stackrel{\mathrm{d}}{=} \Theta(\alpha_1 \oplus \cdots \oplus \alpha_n)$$

et

$$\Phi(\alpha_1, \ldots, \alpha_n) \stackrel{\mathrm{d}}{=} \tau^{-1}(\Phi(\tau(\alpha_1 \oplus \cdots \oplus \alpha_n))).$$

(c) Pour toute $\alpha \in \mathcal{P}art\mathcal{U}n$, tout $m > 0$ et tout $x_1, \ldots, x_m$,

$$\Theta(\alpha, x_1, \ldots, x_m) \stackrel{\mathrm{d}}{=} \lambda x.[\Theta(\alpha)(\langle x_1, \ldots, x_m, x \rangle)]$$

et

$$\Phi(\alpha, x_1, \ldots, x_m) \stackrel{\mathrm{d}}{=} \{(x, y) \mid \langle x_1, \ldots, x_m, x, y \rangle \in \Phi(\tau(\alpha))\}.$$

Notons explicitement que si Φ est un opérateur d'énumération, ψ un SP et α une fonction partielle, $\Phi(\psi)$ et $\Phi(\alpha)$ ne sont en général *pas* des fonctions partielles, mais bien des ensembles, qui ne sont pas nécessairement univalués, de couples. Ainsi, on peut dire qu'un opérateur d'énumération Φ détermine un opérateur récursif Θ ssi pour toute α, $\Phi(\alpha) = \Theta(\alpha)$. Ces conventions de notation peuvent être combinées. Par exemple, pour tout Θ, $\psi, x_1, \ldots, x_m$,

$$\Theta(\psi, x_1, \ldots, x_m) = \lambda x.[\Theta(\widehat{\psi})(\langle x_1, \ldots, x_m, x \rangle)].$$

1.3.8 Séquences finies d'entiers

Nous utiliserons à l'occasion des séquences finies (possiblement vides) d'entiers. La variable σ dénote de telles séquences ; ν dénote la séquence vide. Pour toute σ, on dénote par $\ell(\sigma)$ la *longueur* de σ, i.e., le nombre d'occurrences d'entiers dans σ. On dénote par $\sigma_1, \ldots, \sigma_k$ les éléments d'une séquence σ de longueur k. L'opération de formation de séquences est dénotée par " :". Pour tout x et y, $x : y$ dénote la séquence σ de longueur 2 telle que $\sigma_1 = x$ et $\sigma_2 = y$. Pour toute σ et tout x, $x : \sigma$ (respectivement, $\sigma : x$) dénote la séquence obtenue de σ en la prolongeant par le début (respectivement, par la fin) d'une occurrence de x. L'opération " :" est associative. Clairement, les séquences peuvent être encodées en entiers suivant un schème très simple (voir par exemple [Rog87, § 5.6]). On dira qu'un prédicat sur les séquences est récursif ssi le prédicat correspondant sur les séquences codées est récursif.

1.3.9 Autres conventions de notation

Sauf indication contraire, $\mathcal{C}$ dénote des classes de fonctions partielles et $\mathcal{S}$ dénote des classes de SP. Les noms formés de caractères minuscules sans empattement (rs, $\mathsf{s\text{-}1\text{-}1}$, comp) dénotent des *relations sémantiques* (Définition 2.9). Les noms formés de caractères majuscules sans empattement (A, RIC, RVE) dénotent des ensembles de relations sémantiques.

Chapitre 2

Structures de contrôle, relations sémantiques, relations vérifiables par énumération

Riccardi a introduit dans [Ric80] la notion formelle de *structure de contrôle*. Une structure de contrôle peut être "implantable" ou non dans un SP donné. Intuitivement, une structure de contrôle correspond à une "technique de programmation" (ou technique de preuve), et le fait qu'une structure de contrôle soit implantable dans un SP signifie que la technique correspondante est utilisable.

Dans ce chapitre, après avoir présenté et quelque peu discuté la notion de structure de contrôle, nous montrons que certaines techniques de programmation *naturelles*, et en particulier, la *pseudo-inversion*, ne sont pas représentables par une structure de contrôle. La pseudo-inversion est non seulement très naturelle, elle est aussi utilisable dans tout SP acceptable ; il s'agit donc d'une technique de programmation "portable". Nous proposons alors une notion formelle extrêmement générale de "technique de programmation", la *relation sémantique*, qui nous permet à tout le moins de *définir* la pseudo-inversion, et d'énoncer formellement le fait qu'elle n'est pas représentable par une structure de contrôle.

Après avoir discuté et comparé la notion de relation sémantique avec celle de structure de contrôle, nous introduisons une notion intermédiaire, celle

de *relation vérifiable par énumération*. Nous montrons que la classe des relations vérifiables par énumération est strictement plus grande que celle des structures de contrôle, et contient entre autres la pseudo-inversion.

Nous montrons que les notions *d'extensionnalité* d'une structure de contrôle, définies par Royer [Roy87], sont transposables dans le monde des relations sémantiques, et nous étudions les liens entre les notions de Royer et les notions transposées. Nous obtenons entre autres deux nouvelles caractérisations des structures de contrôle extensionnelles (Théorèmes 2.37 et 2.50). Nous généralisons aussi la notion de "garantie" d'une structure de contrôle par une autre (étudiée par Riccardi mais formellement définie par Royer) aux relations sémantiques.

Nous montrons aussi que la classe des relations vérifiables par énumération conserve certaines propriétés intéressantes de celle des structures de contrôle, entre autres, la propriété de posséder de vastes sous-classes de techniques de programmation "portables", i.e., utilisables dans tous les SP acceptables. Notre résultat sur ce sujet (notre Théorème 2.69) illustre la diversité des techniques de programmation utilisables dans n'importe SP acceptable et, partant, la *similitude* des SP acceptables. Il améliore sensiblement les résultats similaires connus précédemment.

2.1 Structures de contrôle

Nous rappelons au lecteur que nous utilisons certaines conventions de notation relatives aux opérateurs récursifs et d'énumération (Notation 1.16).

2.1 Définition (Riccardi [Ric80, Roy87]) Soit $m > 0$.

(a) Un *schéma de structure de contrôle à m entrées* est un couple (Γ, P), où Γ est un opérateur récursif et P est un prédicat sur les fonctions m-aires.

(b) On dit qu'une fonction (totale) f est une *instance* d'un schéma de structure de contrôle à m-entrées (Γ, P) dans un SP ψ ssi f est m-aire, $P(f)$ est vrai, et pour tout $p_1, \ldots, p_m$,

$$\psi_{f(p_1,\ldots,p_m)} = \Gamma(\psi, p_1, \ldots, p_m, f(p_1, \ldots, p_m)).$$

Si f est récursive, on la dit *instance effective* de (Γ, P) dans ψ.

(c) On dit que S est une *structure de contrôle* à m-entrées ssi il existe (Γ, P), un schéma de structure de contrôle à m-entrées tel que $S = \{(\psi, f) \mid \psi$ est un SP et f est une instance de (Γ, P) dans $\psi\}$. On dit alors que (Γ, P) *détermine* ou encore *est un schéma pour S*.

(d) On dit qu'une fonction (totale) f est une *instance* d'une structure de contrôle S *dans* un SP ψ ssi $(\psi, f) \in S$. Si f est récursive, on la dit *instance effective* de S *dans* ψ.

Souvent le nombre d'entrées d'une structure de contrôle ou d'un schéma de structure de contrôle n'est donné qu'implicitement.

Notons que les définitions correspondantes dans [Roy87] (Définition 1.3.7) ne concernent que les SP *exécutables*. La définition originelle de Riccardi, de même que la nôtre ici, concernent des SP *arbitraires*. La définition de Riccardi, par contre, ne concerne que les SP à puissance de calcul maximale, alors que celle de Royer, comme la nôtre, s'applique à tous les SP, sans restriction sur leur puissance de calcul.

Voici un exemple de schéma de structure de contrôle.

2.2 Exemple $COMP \stackrel{\mathrm{d}}{=} (\Gamma, P)$, où pour tout ψ, p_1, p_2, q,

$$\Gamma(\psi, p_1, p_2, q) = \psi_{p_1} \circ \psi_{p_2},$$

et où, pour toute fonction f 2-aire,

$$P(f) \iff (\forall p_1, p_2)[f(p_1, p_2) \notin \{p_1, p_2\}].$$

Notons que le paramètre q n'apparaît pas dans le membre de droite de l'équation ci-dessus ; nous reviendrons sur ce fait dans quelques instants. Le schéma $COMP$ correspond à la technique de programmation consistant à passer de n'importe quelle paire de programmes à un troisième programme, *distinct des programmes de départ*, calculant la composition fonctionnelle des fonctions partielles calculées par ceux-ci.

Intuitivement, on peut dire que l'opérateur récursif dans un schéma de structure de contrôle représente la *contrainte sémantique* d'une technique de programmation, et que le prédicat représente une *contrainte textuelle* associée à la technique.

En première analyse, on peut visualiser l'opérateur récursif d'un schéma de structure de contrôle comme spécifiant une *transformation sémantique*. En termes de technique de programmation, le rôle de cette transformation sémantique est illustré par l'exemple suivant : un programmeur dans l'exercice de ses fonctions est dans une situation où il possède m programmes $p_1, \ldots, p_m$. Il désire en obtenir un autre dont la sémantique est donnée par une certaine *transformation* de la sémantique des programmes $p_1, \ldots, p_m$. S'il a à sa disposition une instance effective f d'un schéma de structure de contrôle dont l'opérateur récursif correspond à cette transformation, alors, il peut obtenir un programme de sémantique désirée simplement en calculant $f(p_1, \ldots, p_m)$.

Cependant, la définition d'instance d'un schéma de structure de contrôle est beaucoup plus générale que notre exemple ne l'illustre. En particulier, la "sémantique transformée" peut être exprimée en fonction de la sémantique *globale* du SP, et non seulement de celle des programmes $p_1, \ldots, p_m$. De plus, les *textes* des programmes $p_1, \ldots, p_m$ peuvent intervenir dans l'expression de cette "sémantique transformée".

Également, notons que la valeur $f(p_1, \ldots, p_m)$, qui apparaît comme argument à l'opérateur récursif Γ dans la Définition 2.1 (b), correspond au "programme transformé". La présence de ce paramètre, qu'on appelle *paramètre auto-référenciel*, permet d'exprimer des "transformations sémantiques" auto-référencielles, c'est-à-dire où l'expression de la "sémantique transformée" fait intervenir le programme transformé lui-même. Si l'auto-référence est limitée à la *sémantique* du programme transformé, on parle d'auto-référence *extensionnelle*. Si même le *texte* du programme transformé intervient dans l'expression de la "sémantique transformée", on parle d'auto-référence *intensionnelle*. Plusieurs schémas de structures de contrôle (dont l'exemple $COMP$ ci-dessus) "ignorent", en effet, le paramètre auto-référenciel.

Intuitivement, donc, l'opérateur récursif d'un schéma de structure de contrôle ne correspond pas à une simple "transformation sémantique", mais plutôt à un certain "rapport sémantique" qui doit exister entre la sémantique *globale* du SP, les *textes* des programmes de départ et celui du programme transformé. Il demeure toutefois vrai que l'opérateur récursif d'un schéma de structure de contrôle détermine l'aspect *sémantique* de la technique de programmation correspondante.

La contrainte textuelle imposée par le prédicat d'un schéma de structure de

contrôle correspond à une contrainte *globale* sur les *textes* de programmes retournés par les transformations de programmes candidates à "réaliser" la technique de programmation correspondant au schéma. Typiquement, cette contrainte textuelle sera l'injectivité, ou la monotonicité en un argument ou plusieurs. Il n'est pas faux de dire que les contraintes textuelles sont d'abord et avant tout utilisées dans des schémas qui correspondent à des techniques de *preuves* et non vraiment à des techniques de *programmation*.

Très souvent, on ne veut spécifier *aucune* contrainte textuelle. Pour ce faire, on utilisera le prédicat suivant :

2.3 Définition Pour tout $m > 0$, *VRAI* dénote le prédicat sur les fonctions m-aires qui est toujours vrai. Le symbole *VRAI* est surchargé (*"overloaded"*), mais il sera toujours clair d'après le contexte de quel $m > 0$ il s'agit.

Un exemple de schéma de structure de contrôle sans contrainte textuelle serait (Γ, *VRAI*), où Γ est comme dans l'Exemple 2.2. Ce schéma correspond à la propriété de composition effective mentionnée en § 1.1 : un SP est doté de la propriété de composition effective ssi il possède une instance effective de ce schéma. Les propriétés de programmabilité et de substitution effective (également mentionnées en § 1.1), de même que celle correspondant au Théorème de récursion de Kleene (mentionnée en § 1.2), sont aussi exprimables comme schémas de structures de contrôle avec prédicat *VRAI* [Roy87].

Royer a fait remarquer que certaines structures de contrôle étaient déterminées par plusieurs schémas *distincts* (parfois, ayant le même prédicat). Par exemple, la structure de contrôle primitive-recursion [Roy87, Définition 1.3.13] est déterminée par deux schémas à prédicat *VRAI* distincts [Roy87, Définition 2.3.1]. Cet état de choses est vraisemblablement la raison pour laquelle Royer a recours à la notion de structure de contrôle en plus de celle de schéma de structure de contrôle : si deux techniques de programmation (correspondant à deux schémas déterminant la même structure de contrôle) ont exactement les mêmes implantations dans tous les SP, l'objet formel les représentant (la structure de contrôle) est alors unique. Nous adoptons dans ce travail la position qu'il n'y a aucune objection à ce que deux objets formels distincts, chacun représentant intuitivement une technique de programmation, aient exactement les mêmes implantations dans tous les SP. En quelque sorte, nous considérons alors les deux techniques de programmation comme distinctes. Ainsi, nous utiliserons généralement les *schémas* de structure de

contrôle comme représentants formels des techniques de programmation, plutôt que les structures de contrôle déterminées par ceux-ci. De même, la notion de relation sémantique que nous définirons sous peu sera en esprit beaucoup plus près du schéma de structure de contrôle que de la structure de contrôle.

Un des intérêts fondamentaux de la notion de structure de contrôle est qu'elle donne lieu à un résultat de *complétude expressive*, qui laisse entrevoir l'étendue de la versatilité des SP acceptables, i.e., la variété des techniques de programmation qui y sont utilisables. Ce résultat, déjà remarqué par Case dans [Ric80] (p. 52), est exprimé dans sa version la plus forte dans [Roy87] (Théorème 1.3.20). Nous reproduisons ici la partie de ce résultat qui concerne l'étendue de la versatilité des SP acceptables. Nous avons besoin d'une définition préalable.

2.4 Définition (Royer [Roy87]) Un prédicat sur les fonctions m-aires P est dit *prédicat MR* ssi $P(f)$ est vrai pour toute fonction m-aire f injective et strictement croissante en chacun de ses arguments.

2.5 Théorème (Royer [Roy87]) *Pour tout $m > 0$, pour tout opérateur récursif Γ, tout prédicat MR sur les fonctions m-aires P et tout SP acceptable ψ, il y a une instance effective du schéma de structure de contrôle à m entrées (Γ, P) dans ψ.*

Il est clair que le prédicat *VRAI* est MR. Donc, la classe de techniques de programmation visée par le précédent théorème est très vaste et inclut beaucoup de techniques "naturelles", dont celles correspondant à la composition effective, à la substitution effective, à la programmabilité et au Théorème de récursion de Kleene. Cependant, nous montrons maintenant qu'il existe au moins une technique de programmation *naturelle* qui n'est pas exprimable comme structure de contrôle. Nous baptisons cette technique la *pseudo-inversion*.

2.6 Définition Soient α et β des fonctions partielles unaires. On dit que β est une *pseudo-inverse* de α ssi $\mathbf{domaine}(\beta) = \mathbf{image}(\alpha)$ et $\alpha \circ \beta$ est la restriction de l'identité à $\mathbf{domaine}(\beta)$. [1]

1. Alan Selman appelle une pseudo-inverse de α un *raffinement univalué de la relation* α^{-1}.

On dira qu'une fonction i est une *fonction de pseudo-inversion* pour un SP ψ ssi pour tout p, $\psi(i(p))$ est une pseudo-inverse de $\psi(p)$. On dira que la technique de *pseudo-inversion* est utilisable dans un SP ψ ssi il existe une fonction *récursive* de pseudo-inversion pour ψ.

Les deux énoncés qui suivent peuvent être prouvés directement. Cependant, comme ils découlent de résultats ultérieurs, nous ne donnons pas les preuves directes.

2.7 Proposition *La technique de pseudo-inversion est utilisable dans tout SP acceptable.*

Preuve. La proposition découle des Théorèmes 2.53 et 2.69 et de la preuve de la Proposition 2.14. □

2.8 Théorème *Il n'existe aucun schéma de structure de contrôle* (Γ, P) *tel que pour tout* ψ *et toute fonction* i*,* i *est une fonction de pseudo-inversion pour* ψ *ssi* i *est une instance de* (Γ, P) *dans* ψ*.*

Preuve. Le théorème est une conséquence du Corollaire 2.52, du Théorème 2.53 et de la preuve de la Proposition 2.14.

□

2.2 Relations sémantiques

Nous introduisons maintenant le concept de *relation sémantique* (abrégé RS), avec lequel il nous sera possible de définir précisément la pseudo-inversion, et en général, plus de "techniques de programmation" qu'avec la structure de contrôle. D'une certaine façon la relation sémantique est, dans la "lignée" de la structure de contrôle, la version la plus générale possible de la notion de "technique de programmation". Rappelons que $\mathcal{SP}$ dénote l'ensemble de tous les systèmes de programmation.

2.9 Définition Soit $m > 0$.

(a) Une *relation sémantique simple à m entrées* est une relation sur (i.e., un sous-ensemble de) $\mathcal{SP} \times N^{m+1}$.
(b) Une *relation sémantique à m entrées* est un couple $(\mathcal{R}, P)$, où $\mathcal{R}$ est une relation sémantique simple à m-entrées et P est un prédicat sur les fonctions m-aires.
(c) On dit qu'une fonction (totale) f est une *instance* d'une relation sémantique à m entrées $(\mathcal{R}, P)$ dans un SP ψ ssi f est m-aire, $P(f)$ est vrai et pour tout $p_1, \ldots, p_m$,

$$\mathcal{R}(\psi, p_1, \ldots, p_m, f(p_1, \ldots, p_m)).$$

On dit alors aussi que f *satisfait* $(\mathcal{R}, P)$ *en* ψ. Si f est récursive, on la dit *instance effective* de $(\mathcal{R}, P)$ *dans* ψ, et on dit aussi qu'elle *satisfait récursivement* $(\mathcal{R}, P)$ *en* ψ.

Par convention, la locution "relation sémantique" (resp., "relation sémantique simple"), sans spécification d'un nombre d'entrées, signifie "relation sémantique à m entrées, pour un certain $m > 0$" (resp., "relation sémantique simple à m entrées, pour un certain $m > 0$"). À l'occasion, on se permettra de traiter une relation sémantique simple $\mathcal{R}$ comme une RS, il faudra alors comprendre qu'on veut parler de la RS $(\mathcal{R}, \mathit{VRAI})$.

2.10 Conventions Sauf indication contraire :

1. $\mathcal{R}$ dénote des relations sémantiques simples.
2. Les noms formés de lettres minuscules sans empattement (rs, s-1-1, comp) dénotent des RS.
3. Les noms formés de lettres majuscules sans empattement (RIC, RVE) dénotent des ensembles de RS.

Notons que dans [Roy87], les noms formés de lettres sans empattement dénotent des *structures de contrôle*.

On dira qu'une RS est *récursivement satisfaisable dans* un SP ψ ssi il existe une fonction qui la satisfait récursivement en ψ, i.e., ssi il en existe une instance effective en ψ. L'ensemble des RS récursivement satisfaisables dans un SP représente intuitivement l'ensemble des "techniques de programmation" (dans un sens extrêmement large) utilisables dans ce SP.

Comme les schémas de structures de contrôle, les RS ont deux composantes : une *sémantique* (la relation sémantique simple) et l'autre *syntaxique* ou *textuelle* (le prédicat). En termes de transformation de programmes, la relation sémantique simple décrit une relation qui doit exister entre les programmes de départ et le programme image (relation impliquant intuitivement "surtout" la sémantique des programmes, mais pouvant aussi impliquer leur *texte*), et le prédicat décrit une contrainte textuelle (i.e., sur le texte des programmes produits) *globale* (e.g., l'injectivité, la monotonicité) devant être respectée par la transformation de programmes. Une instance d'une RS dans un SP donné est une transformation de programmes qui (i) respecte la contrainte textuelle de la RS, et (ii) en respecte aussi, dans ce SP, la contrainte sémantique.

2.11 Définition Soient a et b chacun une RS ou un schéma de structure de contrôle ou une structure de contrôle. On dit que a et b *coïncident quant aux instances* ssi pour tout $\psi \in \mathcal{SP}$ et toute $f \in \mathcal{T}ot$, f est une instance de a dans ψ ssi f est une instance de b dans ψ.

Pour faire immédiatement le lien avec les structures de contrôle de Riccardi, nous introduisons la définition suivante :

2.12 Définition Une RS $\mathsf{rs} = (\mathcal{R}, P)$ est dite *de Riccardi* ssi il existe un opérateur récursif Θ tel que pour tout $\psi,\ p_1, \ldots, p_m,\ q$,

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow \psi_q = \Theta(\psi, p_1, \ldots, p_m, q).$$

On dit alors que Θ *détermine* rs *à la Riccardi*.

On vérifie facilement, à l'aide des Définitions 2.1 et 2.12, que toute RS de Riccardi coïncide quant aux instances avec une structure de contrôle et que toute structure de contrôle coïncide quant aux instances avec une RS de Riccardi : c'est ce qui motive l'appellation "de Riccardi". [2] (Corollaire : toute technique de programmation exprimable comme structure de contrôle est aussi exprimable comme RS.)

Voici maintenant la définition formelle de la pseudo-inversion.

2. Notons que certaines RS qui ne sont pas de Riccardi peuvent coïncider quant aux instances avec des structures de contrôle.

2.13 Définition ps-inv $\stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout $\psi,\ p,\ q$,

$$\mathcal{R}(\psi, p, q) \Longleftrightarrow \psi_q \text{ est une pseudo-inverse de } \psi_p.$$

2.14 Proposition ps-inv *"représente" bien la pseudo-inversion.*

Preuve. On vérifie facilement que toute fonction totale est une instance de ps-inv dans un SP ssi elle est une fonction de pseudo-inversion (pas nécessairement récursive) dans ce SP.

□

(Corollaire : le formalisme des RS est donc strictement plus expressif que celui des structures de contrôle.)

Le fait que la pseudo-inversion est utilisable dans tout SP acceptable (la Proposition 2.7), s'exprime maintenant en disant que ps-inv est récursivement satisfaisable dans tout SP acceptable, ou encore que tout SP acceptable possède une instance effective de ps-inv. Le Théorème 2.8 énonce que ps-inv ne coïncide quant aux instances avec aucun schéma de structure de contrôle (et donc, aucune structure de contrôle).

Mentionnons tout de suite un autre exemple de l'expressivité accrue des RS par rapport aux structures de contrôle. Une *pseudo-structure de contrôle* [Roy87] est un ensemble de couples (ψ, f) qui n'est déterminé par aucun schéma de structure de contrôle, mais qui présente néanmoins un intérêt. Royer doit avoir recours à une pseudo-structure de contrôle pour représenter la technique de programmation correspondant à la forme point-fixe, due à Rogers, du Théorème de récursion [Rog87, Roy87, Définition 1.4.1.6]. La pseudo-structure de contrôle définie par Royer est la suivante :

$$A \stackrel{\mathrm{d}}{=} \{(\psi, f) \mid \psi \in \mathcal{SP} \ \&\ (\forall p)[\psi_p \text{ totale } \Rightarrow \psi(f(p)) = \psi(\psi_p(f(p)))]\}.$$

Cette pseudo-structure de contrôle peut être représentée sans traitement spécial dans le monde des RS. En effet, on vérifie facilement que $A = \{(\psi, f) \mid f \text{ est une instance de } (\mathcal{R}, \mathit{VRAI}) \text{ dans } \psi\}$, où pour tout $\psi,\ p,\ q$,

$$\mathcal{R}(\psi, p, q) \stackrel{\mathrm{d}}{\Longleftrightarrow} [\psi_p \text{ totale } \Rightarrow \psi(f(p)) = \psi(\psi_p(f(p)))].$$

2.3 Relations vérifiables par énumération

Comme nous avons déjà dit, un des intérêts de la notion de structure de contrôle est qu'elle donne lieu à un résultat sur l'étendue de la versatilité des SP acceptables (le Théorème 2.5, dû à Royer). La RS, étant donnée l'absence totale de restrictions sur la "complexité" ou même la "faisabilité" de la tâche de vérifier les relations sémantiques simples, ne peut dans sa pleine généralité donner lieu à un tel résultat. En effet, pour n'importe quelle classe $\mathcal{S}$ de SP, la RS $(\mathcal{R}, \mathit{VRAI})$, définie par $\mathcal{R}(\psi, p, q) \overset{\mathrm{d}}{\iff} \psi \in \mathcal{S}$ pour tout ψ, p et q, est récursivement satisfaisable précisément dans les SP de $\mathcal{S}$. Si on prend $\mathcal{S} \overset{\mathrm{d}}{=} \mathcal{SP} - \mathcal{SPA}$, on a une RS qui est récursivement satisfaisable précisément dans les SP qui ne sont *pas* acceptables.

Nous introduisons maintenant une classe de RS, celle des RS *vérifiables par énumération*, dont nous montrerons qu'elle est strictement plus grande que celle des RS de Riccardi, *contient en particulier* ps-inv, et donne malgré tout lieu à un résultat sur l'étendue de la versatilité des SP acceptables (un théorème de *complétude expressive*, pour utiliser la terminologie de [Roy87]).

Avant de définir formellement la RS vérifiable par énumération, voici intuitivement comment elle se distingue de la RS de Riccardi. Une RS à m entrées $\mathsf{rs} = (\mathcal{R}, P)$ est de Riccardi ssi il existe un opérateur récursif Γ tel que pour tous $\psi, p_1, \ldots, p_m$ et q, $\mathcal{R}(\psi, p_1, \ldots, p_m, q) \iff \Gamma(\psi, p_1, \ldots, p_m, q) = \psi_q$. Fixons $\psi, p_1, \ldots, p_m$ et q. Intuitivement, on peut dire que $\Gamma(\psi, p_1, \ldots, p_m, q)$ calcule, pour tout x, une "valeur admissible pour x", et que $\mathcal{R}(\psi, p_1, \ldots, p_m, q)$ ssi pour tout x, $\psi_q(x)$ est "admissible pour x". Informellement, si au lieu de faire *calculer* par $\Gamma(\psi, p_1, \ldots, p_m, q)$ une valeur "admissible pour x", on lui fait *vérifier si* une valeur donnée est "admissible pour x" ou non, on obtient la RS vérifiable par énumération. On rajoute le paramètre x à Γ, et on dit que y est "admissible pour x" ssi $\Gamma(\psi, p_1, \ldots, p_m, q, x)(y)\!\downarrow$. Comme pour la RS de Riccardi, on demande que $\mathcal{R}(\psi, p_1, \ldots, p_m, q)$ ssi pour tout x, $\psi_q(x)$ est "admissible pour x". Voilà en essence l'idée intuitive derrière la RS vérifiable par énumération. La définition formelle doit cependant prévoir le cas où $\psi_q(x)\!\uparrow$; également, pour une raison technique dont nous reparlerons bientôt, on fait vérifier l'admissibilité d'une valeur y par $\Gamma(\psi, p_1, \ldots, p_m, q, x)$ en lui soumettant $y + 1$, et non y.

2.15 Définition Soit $m > 0$.

(a) Soient Θ un opérateur récursif et $\psi \in \mathcal{SP}$. On dit que q est *compatible avec* $(\psi, p_1, \ldots, p_m)$ *via* Θ ssi

$$\begin{aligned}(\forall x)[[\psi_q(x)\downarrow \Rightarrow \Theta(\psi, p_1, \ldots, p_m, q, x)(\psi_q(x)+1)\downarrow] \\ \& \ [\psi_q(x)\uparrow \Rightarrow [\Theta(\psi, p_1, \ldots, p_m, q, x)(0)\downarrow \\ \vee\neg(\exists y)[\Theta(\psi, p_1, \ldots, p_m, q, x)(y+1)\downarrow]]]].\end{aligned}$$

(b) On dit qu'un opérateur récursif Θ *détermine* une RS $\mathsf{rs} = (\mathcal{R}, P)$ ssi pour tout $\psi, p_1, \ldots, p_m, q$,

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow q \text{ est compatible avec } (\psi, p_1, \ldots, p_m) \text{ via } \Theta.$$

(c) Une RS $\mathsf{rs} = (\mathcal{R}, P)$ est dite *vérifiable par énumération* ssi il existe un opérateur récursif qui détermine rs.

Une RS vérifiable par énumération est aussi appelée *relation vérifiable par énumération*, que l'on abrège *RVE*.

Encore une fois, l'intuition sous-jacente à la Définition 2.15 est la suivante : le fait que $\Theta(\psi, p_1, \ldots, p_m, q, x)(y+1)\downarrow$ ou non nous dit s'il est admissible ou non, pour un programme q qui "prétend" être en relation $\mathcal{R}$ avec ψ et $p_1, \ldots, p_m$, de retourner y sur entrée x. A priori, plusieurs sorties possibles sont admissibles, et c'est essentiellement la raison pour laquelle la RVE s'avérera plus générale que la RS de Riccardi. [3] $\Theta(\psi, p_1, \ldots, p_m, q, x)$ peut "signifier" l'admissibilité pour q de *diverger* (i.e., ne pas retourner de sortie) sur x soit en convergeant sur 0, soit en divergeant sur $y+1$ pour tout y. La première possibilité est présente pour permettre à $\Theta(\psi, p_1, \ldots, p_m, q, x)$ d'accepter à la fois la divergence *et* la production de certaines sorties comme comportements admissibles de ψ_q sur entrée x.

Nous motivons le nom "relation vérifiable par énumération" par le fait qu'il s'agit (abstraction faite du prédicat) de relations telles que, pour tout $\psi, p_1, \ldots, p_m, q$ et x, on peut vérifier si la valeur de ψ_q sur x est "admissible" en suivant une procédure effective (correspondant à l'opérateur récursif qui détermine la RVE) dont l'entrée est une énumération de la fonction universelle de ψ.

Nous notons immédiatement, pour référence future, les deux propositions suivantes :

3. La RS de Riccardi *peut* admettre un certain nombre de sorties distinctes, par utilisation du paramètre auto-référenciel, mais, intuitivement et informellement, *moins* que la RVE.

2.16 Proposition *Soit $m > 0$. Supposons que Θ soit un opérateur récursif tel que pour tout $\psi, p_1, \ldots, p_m, q$ et x, il existe au plus un y pour lequel $\Theta(\psi, p_1, \ldots, p_m, q, x)(y)\downarrow$. Alors, pour toute RS à m entrées* $\mathsf{rs} = (\mathcal{R}, P)$, *$\Theta$ détermine* rs *ssi pour tout $\psi, p_1, \ldots, p_m, q$,*

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow (\forall x, y)[\Theta(\psi, p_1, \ldots, p_m, q, x)(y+1)\downarrow \Leftrightarrow y = \psi_q(x)].$$

Preuve. Immédiat par la Définition 2.15. □

2.17 Proposition *Soient ψ un SP exécutable et* $\mathsf{rs} = (\mathcal{R}, P)$ *une RVE ; soit de plus Θ un opérateur récursif déterminant* rs. *Supposons que f est une fonction telle que (i) $P(f)$ et (ii) pour tout $p_1, \ldots, p_m$, la fonction partielle récursive $\psi(f(p_1, \ldots, p_m))$ correspond à l'algorithme :*

> **Entrée :** x.
> **Algorithme pour $\psi(f(p_1, \ldots, p_m))$:**
> *Par queue de colombe, calculer $\Theta(\psi, p_1, \ldots, p_m, f(p_1, \ldots, p_m), x)(y+1)$ pour chaque $y \in N$. Dès qu'un résultat est obtenu pour un certain $y_0 + 1$, retourner ce y_0.* □ **Algorithme**

Alors, f est une instance de rs *dans ψ. (Note : f produit des programmes "auto-référenciels".)*

Preuve. Immédiat par la Définition 2.15. □

2.18 Définition (a) $\mathsf{RIC} \stackrel{\mathrm{d}}{=} \{\mathsf{rs} \mid \mathsf{rs}$ est de Riccardi$\}$.

(b) $\mathsf{RVE} \stackrel{\mathrm{d}}{=} \{\mathsf{rs} \mid \mathsf{rs}$ est vérifiable par énumération$\}$.

Le théorème suivant donne une caractérisation des RS de Riccardi en termes de RVE. Nous rappelons au lecteur que la terminologie et la notation associées aux opérateurs récursifs sont présentées en § 1.3.7.

2.19 Théorème *Soient $m > 0$ et* rs *une RS à m entrées. Alors,* $\mathsf{rs} \in \mathsf{RIC}$ *ssi* $\mathsf{rs} \in \mathsf{RVE}$ *et* rs *est déterminée par un opérateur récursif Γ tel que pour tout $\psi, p_1, \ldots, p_m, q$, et x, il existe au plus un y pour lequel $\Gamma(\psi, p_1, \ldots, p_m, q, x)(y)\downarrow$.*

Preuve. Soient $m > 0$ et $\mathsf{rs} = (\mathcal{R}, P)$ une RS à m entrées.

$\Rightarrow$: Supposons que $\mathsf{rs} \in \mathsf{RIC}$. Alors il existe Θ, un opérateur récursif qui détermine rs à la Riccardi, c'est-à-dire tel que pour tout $\psi, p_1, \ldots, p_m, q$,

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow (\forall x, y)[\Theta(\psi, p_1, \ldots, p_m, q)(x) = y \Leftrightarrow \psi_q(x) = y]. \tag{2.1}$$

Par des techniques standard, on exhibe facilement un opérateur récursif Γ tel que pour toute α et tout $p_1, \ldots, p_m, q$,

$$\Gamma(\alpha, p_1, \ldots, p_m, q, x)(y)\downarrow \Longleftrightarrow y = \Theta(\alpha, p_1, \ldots, p_m, q)(x) + 1. \tag{2.2}$$

(À titre exemplaire, nous donnons une construction possible de Γ. Supposons que Θ soit déterminé par l'opérateur d'énumération Φ_a. Alors, Γ est déterminé par Φ_b, où W_b contient précisément les règles "D_u cause $\langle p_1, \ldots, p_m, q, x, y+1, 0\rangle$" telles que "$D_u$ cause $\langle p_1, \ldots, p_m, q, x, y\rangle$" est une règle de W_a.)

Fixons $\psi, p_1, \ldots, p_m$ et q. Clairement, par (2.2), il est le cas que pour tout x, il existe au plus un y pour lequel $\Gamma(\psi, p_1, \ldots, p_m, q, x)(y)\downarrow$. De plus, (2.2) implique que pour tout x et y,

$$\Gamma(\psi, p_1, \ldots, p_m, q, x)(y+1)\downarrow \Longleftrightarrow \Theta(\psi, p_1, \ldots, p_m, q)(x) = y. \tag{2.3}$$

De (2.1) et (2.3), on déduit que

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow (\forall x, y)[\Gamma(\psi, p_1, \ldots, p_m, q, x)(y+1)\downarrow \Leftrightarrow y = \psi_q(x)].$$

Par la Proposition 2.16, on conclut que Γ détermine rs.

$\Leftarrow$: Supposons que $\mathsf{rs} \in \mathsf{RVE}$ et qu'elle est déterminée par Γ, un opérateur récursif tel que pour tout $\psi, p_1, \ldots, p_m, q$ et x, il existe au plus un y pour lequel $\Gamma(\psi, p_1, \ldots, p_m, q, x)(y)\downarrow$. Appelons cette propriété la *propriété spéciale* de Γ. Par des techniques standard, on exhibe facilement un opérateur d'énumération Φ tel que pour toute α, tous $p_1, \ldots, p_m, q, x$ et tout y,

$$(x, y) \in \Phi(\alpha, p_1, \ldots, p_m, q) \Longleftrightarrow \Gamma(\alpha, p_1, \ldots, p_m, q, x)(y+1)\downarrow. \tag{2.4}$$

(À titre exemplaire, nous donnons une construction possible de Φ. Supposons que Γ soit déterminé par l'opérateur d'énumération Φ_b. Alors, Φ est déterminé par W_a, où W_a contient précisément les règles "D_u cause $\langle p_1, \ldots, p_m, q, x, y\rangle$" telles qu'il existe un z pour lequel "D_u cause $\langle p_1, \ldots, p_m, q, x, y+1, z\rangle$" est une règle de W_b.)

Clairement, par la propriété spéciale de Γ, Φ détermine un opérateur récursif. Soit Θ cet opérateur récursif. Fixons $\psi, p_1, \ldots, p_m$ et q. Par (2.4), il est clair que pour tout x et y,

$$\Theta(\alpha, p_1, \ldots, p_m, q)(x) = y \Longleftrightarrow \Gamma(\alpha, p_1, \ldots, p_m, q, x)(y+1)\downarrow. \tag{2.5}$$

D'autre part, par le fait que Γ détermine rs, par sa propriété spéciale et par la Proposition 2.16, on a

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow (\forall x, y)[\Gamma(\psi, p_1, \ldots, p_m, q, x)(y+1)\downarrow \Leftrightarrow y = \psi_q(x)]. \tag{2.6}$$

De (2.5) et (2.6), on déduit que

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow (\forall x, y)[\Theta(\psi, p_1, \ldots, p_m, q)(x) = y \Leftrightarrow \psi_q(x) = y],$$

c'est-à-dire que $\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Leftrightarrow \psi_q = \Theta(\psi, p_1, \ldots, p_m, q)$, et donc que Θ détermine rs à la Riccardi. □

Nous mettons en évidence une partie de l'énoncé du théorème précédent :

2.20 Corollaire $\mathsf{RIC} \subseteq \mathsf{RVE}$.

Toute RS de Riccardi est donc vérifiable par énumération. La constructivité de la preuve du Théorème 2.19 nous donne :

2.21 Corollaire *Soit $\Theta_0, \Theta_1, \ldots$ une énumération standard des opérateurs récursifs (voir [Roy87, § 1.2]). Il existe des fonctions récursives 2-aires f et g telles que pour tout m, toute $\mathsf{rs} \in \mathsf{RIC}$ à m entrées, et tout i, si Θ_i détermine rs à la Riccardi, alors $\Theta_{f(m,i)}$ détermine rs, et si Θ_i détermine rs, alors $\Theta_{g(m,i)}$ détermine rs à la Riccardi.*

Royer a fait remarquer que, même parmi les RS de Riccardi dont le prédicat est $VRAI$, certaines sont déterminées par plus d'un opérateur récursif distinct [Roy87, § 2.3]. Il découle de la preuve de la partie "⇒" du Théorème 2.19 que le même énoncé est vrai pour les RVE. Cependant, la simple inspection de la définition de RVE nous indique que tel doit être le cas : en effet, on vérifie aisément que la même RVE est déterminée par les deux opérateurs récursifs (manifestement distincts) Θ et Γ tels que $\Theta(\psi, p, q, x)(y) = 0$ et $\Gamma(\psi, p, q, x)(y) = 1$ pour tout ψ, p, q, x et y.

2.4 Quelques RVE

Nous donnons dans cette section plusieurs exemples de RVE. La RS qui a suscité toute notre réflexion, ps-inv, est une RVE ; ceci découlera du Théorème 2.53. De ce même théorème et du Corollaire 2.52 découlera le fait qu'elle n'est pas exprimable comme structure de contrôle, et donc pas de Riccardi. Voici d'autres exemples de RS qui sont des RVE non-exprimables comme structures de contrôle. Les preuves que ces RS sont des RVE et qu'elles ne sont pas exprimables comme structures de contrôle seront esquissées peu après celle du Théorème 2.53.

2.22 Définition La RS rech-nb ("recherche non-bornée") est définie comme $(\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p, n et q,

$$\begin{aligned}&\mathcal{R}(\psi,p,q) \Longleftrightarrow\\ &\quad (\forall\langle x,y\rangle)[[\psi_q(\langle x,y\rangle)\downarrow \Rightarrow (\exists x' \geq x)[\psi_p(\langle x',y\rangle) = \psi_q(\langle x,y\rangle)]]\\ &\qquad\quad \&\ [\psi_q(\langle x,y\rangle)\uparrow \Rightarrow \neg(\exists x' \geq x)[\psi_p(\langle x',y\rangle)\downarrow]]].\end{aligned}$$

Intuitivement, une instance de rech-nb dans un SP ψ retourne, sur entrée p, un programme qui, sur entrée $\langle x,y\rangle$, effectue une recherche non-bornée d'un $x' \geq x$ tel que $\psi_p(\langle x',y\rangle)\downarrow$. Par exemple, si r est un instance de rech-nb et si p, sur entrée $\langle x,y\rangle$ retourne un cycle de longueur x dans le graphe y (s'il y en a), alors $r(p)$ retourne, sur entrée $\langle x,y\rangle$, un cycle de longueur *au moins* x dans le graphe y (s'il y en a).

La RS rech-nb est très proche de l'idée intuitive de la "queue de colombe". Mais la vraie idée intuitive de la "queue de colombe" est une recherche non-bornée *à partir de* 0. On peut réaliser cette idée en combinant une instance de rech-nb avec une de s-1-1 (Définition 2.25). Si r est une instance de rech-nb et s une instance de s-1-1 dans un SP ψ, alors, pour tout p, $s(r(p),0)$ est un programme qui, sur entrée y, effectue une recherche non-bornée d'un x tel que $\psi_p(\langle x,y\rangle)\downarrow$.

2.23 Définition majoration $\stackrel{\text{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p et q,

$$\mathcal{R}(\psi,p,q) \Longleftrightarrow (\forall x)[[\psi_q(x)\downarrow \Leftrightarrow \psi_p(x)\downarrow]\ \&\ [\psi_q(x)\downarrow \Rightarrow \psi_q(x) \geq \psi_p(x)]].$$

Une instance de majoration dans un SP ψ retourne, sur entrée p, un programme calculant une fonction partielle de même domaine que ψ_p, et qui majore celle-ci partout sur ce domaine.

Tous les exemples que nous avons donnés de RVE qui ne sont pas de Riccardi ont une "saveur" queue de colombe, en ce sens que toute implantation "naturelle" de ces RS dans un SP s'appuierait sur cette technique, et donc, sur un interprète du SP. Il est cependant facile de montrer qu'aucune de ces RS n'*exige* l'exécutabilité, en ce sens que pour chacune d'elle, il existe un SP *non-exécutable* en possédant une instance effective.

Voici maintenant, un peu en vrac, la définition de plusieurs RVE, toutes de Riccardi celles-là, dont nous aurons besoin au cours de ce travail.

2.24 Définition comp $\stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout $\psi,\ p_1,\ p_2$ et q,

$$\mathcal{R}(\psi, p_1, p_2, q) \iff \psi_q = \psi(p_1) \circ \psi(p_2).$$

La RS comp représente la propriété de composition effective dont nous avons déjà parlé.

2.25 Définition Pour chaque $m > 0$, s-m-1 $\stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout $\psi, p, d_1, \ldots, d_m$ et q,

$$\mathcal{R}(\psi, p, d_1, \ldots, d_m, q) \iff \psi_q = \lambda z.\psi_p(\langle d_1, \ldots, d_m, z\rangle).$$

Le cas particulier $m = 1$ de la définition précédente, s-1-1, correspond à la propriété de substitution effective que nous avons déjà mentionnée. Jones [Jon90] présente des applications de s-1-1 en tant que technique de programmation.

2.26 Définition prog $\stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout $\psi,\ p$ et q,

$$\mathcal{R}(\psi, p, q) \iff \psi_q = \phi_p.$$

Une instance effective de prog dans un SP "compile" les programmes d'un langage standard vers le SP en question. Cette RS correspond donc à la propriété de programmabilité dont nous avons déjà parlé. Rappelons qu'une instance effective de prog est aussi appelée "fonction de programmation".

2.27 Définition $\mathsf{krt} \stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p et q,

$$\mathcal{R}(\psi, p, q) \Longleftrightarrow \psi_q = \lambda z.\psi_p(\langle q, z\rangle).$$

La RS krt correspond au Théorème de récursion de Kleene (notre Théorème 1.8 et [Roy87, p. 214]). Une instance de krt retourne des programmes auto-référenciels, où l'auto-référence n'est pas nécessairement *extensionnelle*, i.e., essentiellement limitée à un appel récursif, mais peut aussi être *intensionnelle*, i.e., mettre en cause le *texte* du programme auto-référenciel.

2.28 Définition Pour chaque $n > 0$, $n\text{-}\mathsf{pkrt} \stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout $\psi, p, x_1, \ldots, x_n$ et q,

$$\mathcal{R}(\psi, p, x_1, \ldots, x_n, q) \Longleftrightarrow \psi_q = \lambda z.\psi_p(\langle q, x_1, \ldots, x_n, z\rangle).$$

Les RS n-pkrt, $n > 0$, sont des versions de krt avec paramètres.

2.29 Définition $\mathsf{remb} \stackrel{\mathrm{d}}{=} (\mathcal{R}, P)$, où pour toute f, $P(f) \Leftrightarrow (\forall p)[f(p) \neq p]$, et où pour tout ψ, p et q,

$$\mathcal{R}(\psi, p, q) \Longleftrightarrow \psi_q = \psi_p.$$

La RS remb correspond à une version très élémentaire de "rembourrage" des programmes, où la seule contrainte sur les programmes rembourrés est qu'ils soient distincts des programmes originels. Elle constitue un exemple d'utilisation d'un prédicat dans une RS pour imposer une contrainte textuelle globale sur les instances.

2.30 Définition La rs remb-inf ("rembourrage infini") est définie comme $(\mathcal{R}, P)$, où pour toute f 2-aire,

$$P(f) \Longleftrightarrow (\forall p, n, n')[f(p, n) \neq p \;\&\; [(n \neq n') \Rightarrow f(p, n) \neq f(p, n')]],$$

et où pour $\mathcal{R}$ est comme dans la définition précédente.

Cette RS correspond à une version un peu plus "exigeante" du rembourrage, telle qu'on peut obtenir une infinité de programmes rembourrés distincts pour n'importe quel programme de départ.

2.31 Définition $\mathsf{acc} \stackrel{\mathrm{d}}{=} (\mathcal{R}, P)$, où pour toute f unaire, $P(f) \iff f$ est surjective, et où $\mathcal{R}$ est comme dans la définition de prog.

Par le Théorème d'isomorphisme de Rogers (Théorème 1.7) et un raisonnement très simple, il est facile de montrer que acc correspond à l'acceptabilité, en ce sens qu'un SP est acceptable ssi il possède une instance effective de acc.

Si $\mathsf{rs} = (\mathcal{R}, P)$, alors $\mathsf{rs\text{-}inj}$ dénote la RS $(\mathcal{R}, Q)$ où pour toute f d'arité appropriée, $Q(f) \stackrel{\mathrm{d}}{\iff} [P(f)$ & f est injective$]$. Ainsi, par exemple, $\mathsf{remb\text{-}inj} = (\mathcal{R}, P)$, où pour toute f 2-aire, $P(f) \iff [f$ est injective & $(\forall p, n)[f(p, n) \neq p]]$, et où $\mathcal{R}$ est comme dans la définition de remb.

Les RS comp, $\mathsf{s}\text{-}m\text{-}\mathsf{1}$, krt, $n\text{-}\mathsf{pkrt}$, remb et $\mathsf{remb\text{-}inf}$ ont toutes été définies comme structures de contrôle par Royer. Royer a *utilisé*, mais non *défini* prog [Roy87, Théorème 1.4.3.16]. Il est clair que acc est de Riccardi, et elle peut donc être exprimée comme structure de contrôle. Il semble que le fait que l'acceptabilité puisse être représentée par une seule structure de contrôle n'ait jamais été remarqué auparavant.

2.5 Extensionnalité

Royer a introduit dans [Roy87] deux notions *d'extensionnalité* d'une structure de contrôle : *l'extensionnalité avec récursion* et *l'extensionnalité sans récursion* (appelée simplement "extensionnalité" par Royer). Ces notions s'avèrent des restrictions très naturelles de la structure de contrôle, dotées de propriétés très intéressantes ; par exemple : les techniques de programmation correspondant aux structures de contrôle extensionnelles (avec ou sans récursion) sont "transmises" par inter-traductions effectives mutuelles entre SP [Roy87, § 2.3]. Également, les structures de contrôle extensionnelles sans récursion sont les seules à être exprimables en sémantique dénotationnelle [Sto77] et dans la théorie des schèmes de programmes [Gre75].

Dans cette section, nous définissons des notions d'extensionnalité dans le contexte des RS, et les étudions dans le contexte des RVE et en relation avec les notions d'extensionnalité de Royer. Nous obtenons entre autres deux nouvelles caractérisations des structures de contrôle extensionnelles sans récursion (Théorèmes 2.37 et 2.50).

Rappelons que pour tout $m > 0$, $\underline{m}$ dénote l'ensemble $\{1, \ldots, m\}$.

2.32 Définition Soit $m > 0$ et soit $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ une RS à m entrées.

(a) Soient $E = \{e_1 < \cdots < e_n\} \subseteq \underline{m}$ et $T = \{t_1 < \cdots < t_{m-n}\} = \underline{m} - E$. Il se peut que E soit vide ou égal à $\underline{m}$. On dit que rs est *extensionnelle en* E ssi il existe une relation R sur $N^{m-n} \times \mathcal{P}\mathit{art}\mathcal{R}\mathit{ec}\mathcal{U}\mathit{n}^{n+1}$ telle que pour tout $\psi, p_1, \ldots, p_m, q$,

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \Longleftrightarrow R(p_{t_1}, \ldots, p_{t_{m-n}}, \psi(p_{e_1}), \ldots, \psi(p_{e_n}), \psi(q))$$

(i.e., intuitivement, ssi la valeur de vérité de $\mathcal{R}(\psi, p_1, \ldots, p_m, q)$ ne dépend jamais que de la *sémantique* des ψ-programmes q et $p_e,\ e \in E$, et de la *valeur* des paramètres $p_t,\ t \in T$).

On dit que R *témoigne de l'extensionnalité de* rs en E.

Si R est univaluée (§ 1.3.2), alors rs est dite *extensionnelle univaluée en* E.

(b) On dit que rs est *extensionnelle* (univaluée) ssi il existe un $E \subseteq \underline{m}$ tel que rs est extensionnelle (univaluée) en E.

(c) On dit que rs est *purement extensionnelle* (univaluée) ssi elle est extensionnelle (univaluée) en $\underline{m}$.

Il est clair qu'une unique relation R témoigne de l'extensionnalité d'une RS extensionnelle. Nous dirons parfois simplement "univaluée" au lieu de "extensionnelle univaluée".

Intuitivement et informellement, une RS à m entrées $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ est extensionnelle ssi ses m entrées peuvent être partitionnées en deux sous-ensembles disjoints E et T tels que pour tout $\psi, p_1, \ldots, p_m, q, x$ et y, l'admissibilité du fait que $\psi_q(x) = y$ pour que $\mathcal{R}(\psi, p_1, \ldots, p_m, q)$ soit vrai, ne dépend que des *fonctions partielles calculées par* les programmes $p_i,\ i \in E$ (i.e., leur sémantique ou *extension*) et du *texte* des paramètres (programmes ou non) $p_j,\ j \in T$. Autrement dit, chaque $\psi, p_1, \ldots, p_m$ détermine un ensemble de sémantiques admissibles, qui ne dépend que des fonctions partielles calculées par les ψ-programmes $p_i,\ i \in E$ et du texte des paramètres $p_j,\ j \in T$, et tel que n'importe quel ψ-programme q dont la sémantique est dans cet ensemble est en relation $\mathcal{R}$ avec ψ et $p_1, \ldots, p_m$.

Les RS extensionnelles correspondent à une idée simple de "relation sémantique", où les éléments qui sont (ou ne sont pas) en relation sont chacun,

soit une sémantique de programme (i.e., une fonction partielle), soit un paramètre dénudé de toute sémantique autre que sa propre valeur. Les RS extensionnelles *univaluées* correspondent à une idée simple de "transformation sémantique", où les "entrées" de la transformation sont chacunes, soit une sémantique de programme, soit un paramètre dénudé de toute sémantique autre que sa propre valeur, et où la "sortie" de la transformation est une sémantique de programme.

Nous donnons maintenant les définitions (traduites dans le langage des RS) des notions de Royer d'extensionnalité pour les structures de contrôle [Roy87, Définitions 2.3.1 et 2.3.2].

2.33 Définition Soit $m > 0$ et soit $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ une RS à m entrées. Soient $E = \{e_1 < \cdots < e_n\} \subseteq \underline{m}$ et $\{t_1 < \cdots < t_{m-n}\} = \underline{m} - E$. Il se peut que E soit vide ou égal à $\underline{m}$.

(a) On dit que rs est *extensionnelle en E avec récursion au sens de Royer* ssi il existe un opérateur récursif Θ tel que pour tout $\psi, p_1, \ldots, p_m, q$,

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \iff \psi(q) = \Theta(\psi(p_{e_1}), \ldots, \psi(p_{e_n}), \psi(q), p_{t_1}, \ldots, p_{t_{m-n}}).$$

(b) On dit que rs est *extensionnelle en E sans récursion au sens de Royer* ssi il existe un opérateur récursif Θ tel que pour tout $\psi, p_1, \ldots, p_m, q$,

$$\mathcal{R}(\psi, p_1, \ldots, p_m, q) \iff \psi(q) = \Theta(\psi(p_{e_1}), \ldots, \psi(p_{e_n}), p_{t_1}, \ldots, p_{t_{m-n}}).$$

Clairement, une RS extensionnelle sans récursion au sens de Royer est aussi extensionnelle avec récursion au sens de Royer. Intuitivement, une RS est extensionnelle avec récursion au sens de Royer ssi elle est déterminée à la Riccardi par un opérateur récursif Γ qui, pour tout $\psi, p_1, \ldots, p_m, q$ et x, peut calculer $\Gamma(\psi, p_1, \ldots, p_m, q)(x)$, sans avoir à recourir au *texte* des programmes q et $p_e,\ e \in E$; et en n'utilisant de ψ que de la sémantique des programmes q et $p_e,\ e \in E$, où E est l'ensemble en lequel la RS est extensionnelle. Si Γ peut même se passer de la sémantique du ψ-programme q, alors la RS est extensionnelle *sans* récursion au sens de Royer.

La notation nécessaire à la discussion de l'extensionnalité dans sa forme la plus générale devient vite prohibitive. Pour garder la notation le plus simple possible, **nous ne considérons dans le reste de cette section que les**

RS à deux entrées. Également, toute RS extensionnelle sera supposée être extensionnelle en $\{1\}$. Le lecteur se convaincra facilement que cette restriction n'entraîne aucune perte de généralité.

2.34 Définition (a) Soient Θ un opérateur récursif, $\alpha, \beta \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$ et $p \in N$. On dit que β est *compatible extensionnellement avec* (α, p) via Θ ssi

$$(\forall x)[[\beta(x)\downarrow \Rightarrow \Theta(\alpha, \beta, p, x)(\beta(x)+1)\downarrow] \\ \& [\beta(x)\uparrow \Rightarrow [\Theta(\alpha, \beta, p, x)(0)\downarrow \\ \vee \neg(\exists y)[\Theta(\alpha, \beta, p, x)(y+1)\downarrow]]]].$$

(b) On dit qu'un opérateur récursif Θ *détermine extensionnellement* une RS (à deux entrées) $\mathsf{rs} = (\mathcal{R}, P)$ ssi pour tout ψ, p_1, p_2, q,

$$\mathcal{R}(\psi, p_1, p_2, q) \iff \psi(q) \text{ est compatible extensionnellement avec } (\psi(p_1), p_2) \text{ via } \Theta.$$

Le théorème suivant exhibe une ressemblance très évidente entre les RVE extensionnelles et la notion d'extensionnalité avec récursion au sens de Royer.

2.35 Théorème *Une RS (à deux entrées)* $\mathsf{rs} = (\mathcal{R}, VRAI)$ *est une RVE extensionnelle (en* $\{1\}$*) ssi il existe un opérateur récursif* Θ *qui détermine extensionnellement* rs.

Preuve. $\Rightarrow$: Soient $\mathsf{rs} = (\mathcal{R}, VRAI)$ une RVE (Définition 2.15) extensionnelle (Définition 2.32) et Γ un opérateur récursif déterminant rs. Pour toutes $\alpha, \beta \in \mathcal{P}art\mathcal{U}n$, on définit $\alpha \oslash \beta \stackrel{\mathrm{d}}{=} \lambda\langle i, z\rangle.[\alpha(z)$ si $i = 0$; $\beta(z)$ si $i = 1$; $\uparrow$ autrement$]$. Notons que si α et β sont toutes deux partielles récursives, alors $\alpha \oslash \beta$ est la fonction universelle (unaire) d'un SP dont le programme 0 calcule α et le programme 1 calcule β. Par des techniques standard, on exhibe facilement un opérateur récursif Θ tel que pour toutes α, β, p et x,

$$\Theta(\alpha, \beta, p, x) = \Gamma(\alpha \oslash \beta, 0, p, 1, x).$$

Fixons ψ, p_1, p_2 et q. Comme rs est extensionnelle, pour tout ψ', p'_1 et q' tels que $\psi'(p'_1) = \psi(p_1)$ et $\psi'(q') = \psi(q)$, on a $\mathcal{R}(\psi', p'_1, p_2, q) \iff \mathcal{R}(\psi, p_1, p_2, q)$. En particulier, $\mathcal{R}(\rho, 0, p_2, 1) \iff \mathcal{R}(\psi, p_1, p_2, q)$, où $\widehat{\rho} = \psi(p_1) \oslash \psi(p_2)$.

Puisque $\Theta(\psi(p_1), \psi(q), p_2, x) = \Gamma(\rho, 0, p_2, 1, x)$ pour tout x, on vérifie par la Définition 2.15 que $\mathcal{R}(\rho, 0, p_2, 1)$ ssi $\psi(q)$ est compatible extensionnellement avec $(\psi(p_1), p_2)$ via Θ. On a donc $\mathcal{R}(\psi, p_1, p_2, q) \iff \psi(q)$ est compatible extensionnellement avec $(\psi(p_1), p_2)$ via Θ. On déduit que Θ détermine extensionnellement rs.

$\Leftarrow$: Soit $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ une RS déterminée extensionnellement par un opérateur récursif Θ. Par des techniques standard, on exhibe facilement un opérateur récursif Γ tel que pour tout ψ, p_1, p_2, q et x,

$$\Gamma(\psi, p_1, p_2, q, x) = \Theta(\psi(p_1), \psi(q), p_2, x).$$

On vérifie que Γ détermine rs, et donc que rs est vérifiable par énumération.

Définissons maintenant R par

$$R(p, \alpha, \beta) \iff \beta \text{ est compatible extensionnellement avec } (\alpha, p) \text{ via } \Theta$$

pour tout p et toutes $\alpha, \beta \in \mathcal{P}\mathit{art}\mathcal{R}\mathit{ec}\mathcal{U}\mathit{n}$. Fixons ψ, p_1, p_2 et q. Par la Définition 2.34, $\mathcal{R}(\psi, p_1, p_2, q)$ ssi $\psi(q)$ est compatible extensionnellement avec $(\psi(p_1), p_2)$ via Θ. On a donc $\mathcal{R}(\psi, p_1, p_2, q) \iff R(p_2, \psi(p_1), \psi(q))$. On déduit que R témoigne de l'extensionnalité de rs, et que donc, rs est extensionnelle.

□

Une variante de la preuve du Théorème 2.35 fournit le théorème suivant, une caractérisation de la notion d'extensionnalité avec récursion de Royer : elle coïncide avec notre notion d'extensionnalité restreinte aux RS de Riccardi.

2.36 Théorème *Une RS (à deux entrées) est extensionnelle (en* $\{1\}$*) avec récursion au sens de Royer ssi elle est de Riccardi et extensionnelle (en* $\{1\}$*).*

Preuve. $\Rightarrow$: Soient $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ une RS déterminée extensionnellement avec récursion au sens de Royer (Définition 2.33 (a)) par un opérateur récursif Θ. Par des techniques standard, on exhibe facilement un opérateur récursif Γ tel que pour tout ψ, p_1, p_2 et q,

$$\Gamma(\psi, p_1, p_2, q) = \Theta(\psi(p_1), \psi(q), p_2).$$

Par la Définition 2.33 (a), on a alors $\mathcal{R}(\psi, p_1, p_2, q) \iff \psi(q) = \Gamma(\psi, p_1, p_2, q)$. On déduit que Γ détermine rs à la Riccardi, et que donc, rs est de Riccardi.

Définissons maintenant R par

$$R(p,\alpha,\beta) \Longleftrightarrow \beta = \Theta(\alpha,\beta,p)$$

pour tout p et toutes $\alpha, \beta \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$. Fixons ψ, p_1, p_2 et q. Par la Définition 2.33 (a), $\mathcal{R}(\psi, p_1, p_2, q)$ ssi $\psi(q) = \Theta(\psi(p_1), \psi(q), p_2)$. On a donc $\mathcal{R}(\psi, p_1, p_2, q) \Longleftrightarrow R(p_2, \psi(p_1), \psi(q))$. On déduit que R témoigne de l'extensionnalité de rs, et que donc, rs est extensionnelle.

$\Leftarrow$: Soient $\mathsf{rs} = (\mathcal{R}, VRAI)$ une RS extensionnelle (Définition 2.32) déterminée à la Riccardi par un opérateur récursif Γ (Définition 2.12). Soit $\oslash$ comme dans la preuve du Théorème 2.35. Par des techniques standard, on exhibe facilement un opérateur récursif Θ tel que pour toutes α, β, p et x,

$$\Theta(\alpha,\beta,p) = \Gamma(\alpha \oslash \beta, 0, p, 1).$$

Fixons ψ, p_1, p_2 et q. Comme rs est extensionnelle, pour tout ψ', p_1' et q' tels que $\psi'(p_1') = \psi(p_1)$ et $\psi'(q') = \psi(q)$, on a $\mathcal{R}(\psi', p_1', p_2, q) \Longleftrightarrow \mathcal{R}(\psi, p_1, p_2, q)$. En particulier, $\mathcal{R}(\rho, 0, p_2, 1) \Longleftrightarrow \mathcal{R}(\psi, p_1, p_2, q)$, où $\widehat{\rho} = \psi(p_1) \oslash \psi(p_2)$. Puisque $\Theta(\psi(p_1), \psi(q), p_2) = \Gamma(\rho, 0, p_2, 1)$, et par le fait que Γ détermine rs à la Riccardi, on vérifie que que $\mathcal{R}(\rho, 0, p_2, 1)$ ssi $\psi(q) = \Theta(\psi(p_1), \psi(q), p_2)$. On a donc $\mathcal{R}(\psi, p_1, p_2, q) \Longleftrightarrow \psi(q) = \Theta(\psi(p_1), \psi(q), p_2)$. On déduit que Θ détermine rs extensionnellement avec récursion au sens de Royer, et que donc, rs est extensionnelle avec récursion au sens de Royer. □

Voici une caractérisation des RS extensionnelles *sans* récursion au sens de Royer (donc, aussi, des *structures de contrôle extensionnelles*).

2.37 Théorème *Une RS (à deux entrées) est extensionnelle sans récursion au sens de Royer ssi elle est une RVE extensionnelle univaluée.*

Preuve. $\Rightarrow$: Soit rs une RS extensionnelle sans récursion au sens de Royer. Comme rs est clairement aussi extensionnelle *avec* récursion au sens de Royer, elle est, par le Théorème 2.36, de Riccardi et, par le Corollaire 2.20, une RVE. D'autre part, aussi par le Théorème 2.36, elle est extensionnelle. Il reste à montrer qu'elle est extensionnelle univaluée.

Soit R la relation sur $N \times \mathcal{P}art\mathcal{R}ec\mathcal{U}n^2$ témoignant de l'extensionnalité de rs. Par la Définition 2.33 (b), il existe Θ, un opérateur récursif tel que pour tout $\psi,\ p_1,\ p_2$ et q,

$$\mathcal{R}(\psi, p_1, p_2, q) \Longleftrightarrow \psi(q) = \Theta(\psi(p_1), p_2).$$

D'autre part, comme R témoigne de l'extensionnalité de rs, on sait que pour tout ψ, p_1, p_2 et q,

$$\mathcal{R}(\psi, p_1, p_2, q) \iff R(p_2, \psi(p_1), \psi(q)).$$

On a donc $R(p_2, \psi(p_1), \psi(q)) \iff \psi(q) = \Theta(\psi(p_1), p_2)$. Clairement, donc, R est univaluée, et rs aussi.

$\Leftarrow$: Soit rs une RVE extensionnelle univaluée. Soient Γ un opérateur récursif déterminant rs extensionnellement et R la relation univaluée témoignant de l'extensionnalité de rs.

Dans le reste de cette preuve, α, β et γ dénotent des fonctions finies, et ζ et η dénotent des fonctions partielles récursives. Ces conventions s'appliquent aussi aux symboles décorés. Le symbole *UNIVAL* dénote une opération sur les ensembles finis de couples telle que pour tout tel ensemble A, $\alpha \stackrel{\mathrm{d}}{=} \mathit{UNIVAL}(A)$ est une fonction finie (i.e., un ensemble fini univalué de couples) satisfaisant $\alpha \subseteq A$ et $\mathbf{dom}(\alpha) = \{x \mid (\exists y)[(x, y) \in A]\}$. On suppose *UNIVAL récursive* en termes d'indices canoniques. Clairement, une telle opération existe.

Par les définitions pertinentes, on a que pour toute ζ, η et p,

$$\begin{aligned} R(p, \zeta, \eta) \iff (\forall x)[&[\eta(x)\downarrow \Rightarrow \Gamma(\zeta, \eta, p, x)(\eta(x) + 1)\downarrow] \\ &\& \; [\eta(x)\uparrow \Rightarrow [\Gamma(\zeta, \eta, p, x)(0)\downarrow \\ &\quad \vee \neg(\exists y)[\Gamma(\zeta, \eta, p, x)(y + 1)\downarrow]]]], \end{aligned} \tag{2.7}$$

et il suffit d'exhiber un opérateur récursif Θ tel que pour toutes ζ, η et tout p,

$$R(p, \zeta, \eta) \iff \eta = \Theta(\zeta, p). \tag{2.8}$$

Nous décrivons un opérateur d'énumération Ψ et montrons qu'il détermine un opérateur récursif Θ satisfaisant (2.8).

L'expression "$\Gamma(\alpha, \beta, p, x)(y + 1)\downarrow$ en au plus i étapes" signifie qu'une règle "D_u cause $\langle p, x, y{+}1, z\rangle$" pour un certain z et un certain $D_u \subseteq \alpha \oplus \beta$ apparaît dans les i premières étapes d'une énumération standard de W_a, où Φ_a est l'opérateur d'énumération déterminant Γ. Clairement, par monotonicité des opérateurs d'énumération, si $\Gamma(\alpha, \beta, p, x)(y + 1)\downarrow$ en au plus i étapes, alors pour toute $\gamma \supseteq \beta$ et tout $j \geq i$, $\Gamma(\alpha, \gamma, p, x)(y + 1)\downarrow$ en au plus j étapes. Clairement aussi, l'ensemble $\{\langle x, y\rangle \mid \Gamma(\alpha, \beta, p, x)(y{+}1)\downarrow$ en au plus i étapes$\}$

est fini, et son indice canonique peut être calculé uniformément en i, p et des indices canoniques pour α et β.

Pour chaque α, γ, p et i, définissons

$$E(\alpha,\gamma,p,i) \stackrel{\mathrm{d}}{=} \{(x,y) \mid x \notin \mathbf{dom}(\gamma) \ \&\ \Gamma(\alpha,\gamma,p,x)(y+1){\downarrow} \text{ en au plus } i \text{ étapes}\}.$$

Pour chaque α, β, p et i, définissons $F(\alpha,\beta,p,i)$ récursivement comme suit :

$$\begin{aligned} F(\alpha,\beta,p,0) &\stackrel{\mathrm{d}}{=} \beta, \\ F(\alpha,\beta,p,i+1) &\stackrel{\mathrm{d}}{=} F(\alpha,\beta,p,i) \cup \mathit{UNIVAL}\,(E(\alpha,F(\alpha,\beta,p,i),p,i)). \end{aligned}$$

On définit $F(\alpha,\beta,p) \stackrel{\mathrm{d}}{=} \bigcup_{i\geq 0} F(\alpha,\beta,p,i)$; et aussi $G(\alpha,p,i) \stackrel{\mathrm{d}}{=} F(\alpha,\emptyset,p,i)$ et $G(\alpha,p) \stackrel{\mathrm{d}}{=} F(\alpha,\emptyset,p)$.

On note :

2.38 Faits Pour toutes α, β, γ, tout p et tout i,

1. $E(\alpha,\gamma,p,i) \subseteq E(\alpha,\gamma,p,i+1)$.
2. $F(\alpha,\beta,p,i) \subseteq F(\alpha,\beta,p,i+1)$.
3. $G(\alpha,p,i) \subseteq G(\alpha,p,i+1)$.
4. $E(\alpha,\gamma,p,i)$, $F(\alpha,\beta,p,i)$ et $G(\alpha,p,i)$ sont des ensembles finis de couples.
5. $F(\alpha,\beta,p,i)$ et $G(\alpha,p,i)$ sont des fonctions finies.
6. $F(\alpha,\beta,p)$ et $G(\alpha,p)$ sont des fonctions partielles récursives.

Pour toute ζ, toute η et tout p, nous disons que η est *pré-compatible* avec (ζ,p) ssi

$$(\forall x)[\eta(x){\downarrow} \Rightarrow \Gamma(\zeta,\eta,p,x)(\eta(x)+1){\downarrow}].$$

Pour toute ζ, toute η et tout p, nous disons que η est *compatible* avec (ζ,p) ssi η est pré-compatible avec (ζ,p) et

$$(\forall x)[\eta(x){\uparrow} \Rightarrow \neg(\exists y)[\Gamma(\zeta,\eta,p,x)(y+1){\downarrow}]].$$

Il faut se garder de confondre cette notion de compatibilité (locale à cette preuve) avec celle de la Définition 2.15.

2.39 Lemme *Pour toutes ζ, η et p, si η est compatible avec (ζ, p), alors $R(p, \zeta, \eta)$.*

Preuve. Immédiat par la définition de compatibilité et (2.7).

□ **Lemme 2.39**

Donc, comme R est univaluée, au plus une fonction partielle récursive est compatible avec (ζ, p) pour toute ζ et tout p.

Nous pouvons déjà prouver :

2.40 Lemme *Pour toutes α, β, γ, p et i :*

(a) *Si β est pré-compatible avec (α, p), alors $F(\alpha, \beta, p) \supseteq \beta$ est compatible avec (α, p).*
(b) *Si β et γ sont toutes deux pré-compatibles avec (α, p), alors $\beta \cup \gamma$ est univalué.*
(c) *Si $\alpha \subseteq \alpha'$ et β est pré-compatible avec (α, p), alors β est pré-compatible avec (α', p)*
(d) *$G(\alpha, p, i)$ est pré-compatible avec (α, p).*
(e) *Si $\alpha \subseteq \alpha'$, alors $G(\alpha, p, i) \subseteq G(\alpha', p, i)$.*

Preuve. (c) : Immédiat par définition de pré-compatiblité et monotonicité des opérateurs récursifs.

(d) : Par induction sur i. Base ($i = 0$) : $\emptyset$ est clairement pré-compatible avec (α, p). Pas : Supposons $G(\alpha, p, i)$ pré-compatible avec (α, p). Il suffit de montrer que pour tout $(x, y) \in G(\alpha, p, i+1) - G(\alpha, p, i)$, $\Gamma(\alpha, G(\alpha, p, i+1), p, x)(y+1)\downarrow$, qui est une conséquence de la définition de $G(\alpha, p, i+1)$, du fait que $G(\alpha, p, i) \subseteq G(\alpha, p, i+1)$ et de la monotonicité des opérateurs récursifs.

(a) : Supposons β pré-compatible avec (α, p). Posons $\zeta \overset{\mathrm{d}}{=} F(\alpha, \beta, p)$. De façon similaire à la preuve de (d), on établit que pour tout i, $F(\alpha, \beta, p, i)$ est pré-compatible avec (α, p). Puisque $\zeta = \bigcup_{i \geq 0} F(\alpha, \beta, p, i)$, on déduit que ζ est pré-compatible avec (α, p). Il reste à montrer que pour tout x, $\zeta(x)\uparrow \Rightarrow \neg(\exists y)\Gamma(\alpha, \zeta, p, x)(y+1)\downarrow$. Supposons le contraire, c'est-à-dire qu'il existe x_0, y_0 et k tels que $\zeta(x_0)\uparrow$ et $\Gamma(\alpha, \zeta, p, x_0)(y_0+1)\downarrow$ en au plus k étapes. Par

continuité des opérateurs récursifs, il existe $\gamma \subseteq \zeta$ telle que $\Gamma(\alpha, \gamma, p, x_0)(y_0 + 1)\downarrow$ en au plus k étapes. Comme $\gamma \subseteq \zeta$ et par définition de $F(\alpha, \beta, p)$, il existe $j \geq k$ tel que $\gamma \subseteq F(\alpha, \beta, p, j)$. Par monotonicité des opérateurs récursifs, $\Gamma(\alpha, F(\alpha, \beta, p, j), p, x_0)(y_0 + 1)\downarrow$ en au plus j étapes. Clairement, puisque $x_0 \notin \mathbf{dom}(\zeta)$, $x_0 \notin \mathbf{dom}(F(\alpha, \beta, p, j))$. Alors, par définition de $F(\alpha, \beta, p, j+1)$, $x_0 \in \mathbf{dom}(F(\alpha, \beta, p, j+1))$, d'où, puisque $\zeta = \bigcup_{i \geq 0} F(\alpha, \beta, p, i)$, $x_0 \in \mathbf{dom}(\zeta)$: une contradiction.

(b) : Soient β et γ toutes deux pré-compatibles avec (α, p). Supposons que $\beta \cup \gamma$ n'est pas univalué. Alors, par (a), $F(\alpha, \beta, p)$ et $F(\alpha, \gamma, p)$ sont deux fonctions partielles récursives distinctes (l'une contenant β, l'autre γ) compatibles avec (α, p). Par le Lemme 2.39, cela contredit le fait que R est univaluée.

(e) : Soient α et α' telles que $\alpha \subseteq \alpha'$. Procédons par contradiction. Supposons qu'il existe i tel que $G(\alpha, p, i) - G(\alpha', p, i) \neq \emptyset$, et soit $i \stackrel{\mathrm{d}}{=} (\mu i)[G(\alpha, p, i) - G(\alpha', p, i) \neq \emptyset]$. Clairement, $i > 0$. Soit $(x, y) \in G(\alpha, p, i) - G(\alpha', p, i)$. On note que $(x, y) \notin G(\alpha', p, i-1)$, et donc, $(x, y) \notin G(\alpha, p, i-1)$. Par (d) et (c), on a que $G(\alpha, p, i)$ et $G(\alpha', p, i)$ sont toutes deux pré-compatibles avec (α', p). Par (b), alors, $x \notin \mathbf{dom}(G(\alpha', p, i))$. D'autre part, comme $(x, y) \in G(\alpha, p, i) - G(\alpha, p, i-1)$, on sait que $\Gamma(\alpha, G(\alpha, p, i-1), p, x)(y+1)\downarrow$ en au plus $i-1$ étapes. Comme $\alpha \subseteq \alpha'$ et que $G(\alpha, p, i-1) \subseteq G(\alpha', p, i-1)$, on déduit par monotonicité des opérateurs récursifs que $\Gamma(\alpha', G(\alpha', p, i-1), p, x)(y+1)\downarrow$ en au plus $i-1$ étapes, d'où on tire, par définition de $G(\alpha', p, i)$ et par le fait que $x \notin \mathbf{dom}(G(\alpha', p, i-1))$, que $x \in \mathbf{dom}(G(\alpha', p, i))$, une contradiction.

□ **Lemme 2.40**

L'opérateur d'énumération Ψ est défini par : $\Psi = \Phi_z$, où W_z est tel que pour toute α, tout p, et toute paire $(x, y) \in G(\alpha, p)$, "α cause $\langle p, x, y \rangle$" est une règle de W_z.

2.41 Lemme (a) *Pour toute α et tout p, $\Psi(\alpha, p) = G(\alpha, p)$.*
(b) *Ψ détermine un opérateur récursif.*

Preuve. (a) : Immédiat par la définition de Ψ et le Lemme 2.40 (e).

(b) : Supposons le contraire. Alors, par continuité des opérateurs d'énumération, il existe α et p tels que $\Psi(\alpha, p)$ n'est pas univalué. Or, par (a), $\Psi(\alpha, p) = G(\alpha, p)$, une fonction partielle récursive : contradiction.

□ **Lemme 2.41**

Appelons Θ l'opérateur récursif déterminé par Ψ. Le lemme suivant complète la preuve du théorème.

2.42 Lemme *Θ satisfait (2.8), i.e., pour toutes ζ, η et tout p, $R(p,\zeta,\eta) \iff \eta = \Theta(\zeta,p)$.*

Preuve. Par le fait que R est univaluée, il suffit de montrer que pour toute ζ et tout p, $R(p,\zeta,\Theta(\zeta,p))$. Par le Lemme 2.39, il suffit encore de montrer que pour toute ζ et tout p, $\Theta(\zeta,p)$ est compatible avec (ζ,p).

Fixons ζ et p, et soit $\eta \stackrel{\mathrm{d}}{=} \Theta(\zeta,p)$. Nous montrons d'abord que η est pré-compatible avec (ζ,p). Soient x et y tels que $\eta(x) = y$. Par définition d'opérateur d'énumération, il existe $\alpha \subseteq \zeta$ tel que $(x,y) \in \Theta(\alpha,p)$. Par le Lemme 2.41, $(x,y) \in G(\alpha,p)$. Par définition de $G(\alpha,p)$, il doit exister i tel que $(x,y) \in G(\alpha,p,i)$. Clairement, $(x,y) \notin G(\alpha,p,0) = \emptyset$; soit donc $i \stackrel{\mathrm{d}}{=} (\mu i)[(x,y) \in G(\alpha,p,i)]$. Comme $(x,y) \in G(\alpha,p,i) - G(\alpha,p,i-1)$, on déduit par définition de $G(\alpha,p,i)$ que $\Gamma(\alpha,G(\alpha,p,i-1),p,x)(y+1)\!\downarrow$ en au plus $i-1$ étapes. Comme $\alpha \subseteq \zeta$ et que $G(\alpha,p,i-1) \subseteq G(\alpha,p) = \Theta(\alpha,p) \subseteq \Theta(\zeta,p) = \eta$, on déduit par monotonicité des opérateurs récursifs que $\Gamma(\zeta,\eta,p,x)(y+1)\!\downarrow$.

Il reste à montrer que pour tout x, $\eta(x)\!\uparrow \Rightarrow \neg(\exists y)[\Gamma(\zeta,\eta,p,x)(y+1)\!\downarrow]$. Supposons le contraire, c'est-à-dire qu'il existe x_0, y_0 et k tels que $\eta(x_0)\!\uparrow$ et $\Gamma(\zeta,\eta,p,x_0)(y_0+1)\!\downarrow$ en au plus k étapes. Par continuité des opérateurs récursifs, il existe $\alpha \subseteq \zeta$ et $\beta \subseteq \eta$ telles que $\Gamma(\alpha,\beta,p,x_0)(y_0+1)\!\downarrow$ en au plus k étapes. Comme β est finie et incluse dans $\eta = \Theta(\zeta)$, il existe par continuité des opérateurs récursifs $\gamma \subseteq \zeta$ telle que $\Theta(\gamma) \supseteq \beta$. Posons $\alpha' \stackrel{\mathrm{d}}{=} \alpha \cup \gamma$. Par monotonicité des opérateurs récursifs, $\Gamma(\alpha',\beta,p,x_0)(y_0+1)\!\downarrow$ en au plus k étapes. Comme β est finie et incluse dans $\Theta(\gamma)$, il existe, par le Lemme 2.41 (a) et la définition de $G(\alpha',p)$, $i \geq k$ tel que $\beta \subseteq G(\alpha',p,i)$. Par monotonicité des opérateurs récursifs, on sait que $\Gamma(\alpha',G(\alpha',p,i),p,x_0)(y_0+1)\!\downarrow$ en au plus i étapes. Comme $x_0 \notin \mathbf{dom}(\eta)$, et parce que $G(\alpha',p,i) \subseteq \Theta(\gamma) \subseteq \Theta(\zeta) = \eta$, on sait que $x_0 \notin \mathbf{dom}(G(\alpha',p,i))$. Par la définition de $G(\alpha',p,i+1)$, on a donc que $x_0 \in \mathbf{dom}(G(\alpha',p,i+1))$, d'où $x_0 \in \mathbf{dom}(\eta)$, une contradiction. □ **Lemme 2.42**

□ **Théorème 2.37**

Un commentaire informel sur les définitions de E et F dans la partie "$\Leftarrow$" de

la preuve précédente : même si rs est univaluée, il n'est pas nécessairement le cas que pour toute α et β telles que β est pré-compatible avec α, et tout p et x, il y ait au plus un y tel que $\Gamma(\alpha, \gamma, p, x)(y+1)\downarrow$. En effet, Γ peut "s'amuser" à converger sur autant de valeurs de $y+1$ qu'il lui plaît un fois qu'il a "constaté" que $\gamma(x)$ a convergé. Nous avons donc besoin du test "$x \notin \mathbf{dom}(\gamma)$" dans la définition de E, non seulement pour établir la partie (a) du Lemme 2.40, mais aussi pour que Ψ détermine bien un opérateur récursif. Par contre, on peut montrer que l'utilisation de $UNIVAL$ dans la définition de F n'est nécessaire que pour établir la partie (a) du Lemme 2.40 et la partie (b) du Lemme 2.41.

Une interprétation intuitive de la partie "$\Leftarrow$" de l'énoncé du Théorème 2.37 est que la puissance expressive additionnelle du formalisme des RVE par rapport à celui de la structure de contrôle s'estompe si on se restreint aux RS extensionnelles *univaluées*.

Nous avons le corollaire suivant.

2.43 Corollaire *Une RS extensionnelle avec récursion au sens de Royer est extensionnelle* sans *récursion au sens de Royer ssi elle est univaluée.*

Preuve. $\Rightarrow$: Par la partie "$\Rightarrow$" du théorème, *toute* RS extensionnelle sans récursion au sens de Royer est extensionnelle univaluée.

$\Leftarrow$: Si une RS est extensionnelle avec récursion au sens de Royer, alors par le Théorème 2.36, elle est de Riccardi, et donc, par le Corollaire 2.20, une RVE. Par la partie "$\Leftarrow$" du théorème, si elle est en plus univaluée, alors elle est extensionnelle sans récursion au sens de Royer. □

Le Théorème 2.35 suggère tout naturellement la définition suivante, en contrepartie de la définition d'extensionnalité sans récursion au sens de Royer (Définition 2.33 (b)).

2.44 Définition (a) Soient Θ un opérateur récursif, $\alpha, \beta \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$ et $p \in N$. On dit que β est *compatible extensionnellement sans récursion avec* (α, p) *via* Θ ssi

$$(\forall x)[[\beta(x)\downarrow \Rightarrow \Theta(\alpha, p, x)(\beta(x)+1)\downarrow] \\ \& [\beta(x)\uparrow \Rightarrow [\Theta(\alpha, p, x)(0)\downarrow \\ \vee \neg(\exists y)[\Theta(\alpha, p, x)(y+1)\downarrow]]]].$$

(b) On dit qu'un opérateur récursif Θ *détermine extensionnellement sans récursion* une RS (à deux entrées) $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ ssi pour tout ψ, p_1, p_2, q,

$$\mathcal{R}(\psi, p_1, p_2, q) \iff \psi(q) \text{ est compatible extensionnellement sans récursion avec } (\psi(p_1), p_2) \text{ via } \Theta.$$

(c) Une RS est dite *extensionnelle sans récursion* ssi il existe un opérateur récursif qui la détermine extensionnellement sans récursion.

On vérifie facilement que :

2.45 Observation Toute RS extensionnelle sans récursion est extensionnelle avec récursion, et donc, par le Théorème 2.35, une RVE extensionnelle.

L'extensionnalité, lorsque restreinte aux RS de Riccardi, coïncide avec l'extensionnalité avec récursion au sens de Royer (Théorème 2.36). Par contraste, nous montrons maintenant que l'extensionnalité *sans récursion* restreinte aux RS de Riccardi ne coïncide *pas* avec l'extensionnalité sans récursion au sens de Royer (Définition 2.33 (b)).

2.46 Proposition (a) *Toute RS extensionnelle sans récursion au sens de Royer est de Riccardi et extensionnelle sans récursion.*

(b) *Il existe une RS de Riccardi extensionnelle sans récursion qui n'est pas extensionnelle sans récursion au sens de Royer.*

Preuve. (a) : Soit $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ une RS extensionnelle sans récursion au sens de Royer. Comme rs est clairement aussi extensionnelle *avec* récursion au sens de Royer, alors par le Théorème 2.36, elle est de Riccardi. D'autre part, par la Définition 2.33 (b), il existe Θ, un opérateur récursif tel que pour tout $\psi,\ p_1,\ p_2$ et q,

$$\mathcal{R}(\psi, p_1, p_2, q) \iff \psi(q) = \Theta(\psi(p_1), p_2).$$

Par une construction similaire à celle de la preuve de la partie "$\Rightarrow$" du Théorème 2.19, on montre l'existence de Γ, un autre opérateur récursif tel que pour tout $\psi,\ p_1,\ p_2$ et q,

$$\mathcal{R}(\psi, p_1, p_2, q) \iff (\forall x, y)[\Gamma(\psi(p_1), p_2, x)(y+1)\downarrow \Leftrightarrow y = \psi_q(x)],$$

et tel que pour toute α, tout p et tout x, il existe au plus un y pour lequel $\Gamma(\alpha, p, x)(y)\downarrow$. Par une légère variante de la Proposition 2.16, on a alors que Γ détermine rs extensionnellement sans récursion.

(b) : Soit $\mathsf{bidon} \stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p_1, p_2 et q, $\mathcal{R}(\psi, p_1, p_2, q)$ est vrai. On vérifie facilement que rs est déterminée extensionnellement sans récursion par l'opérateur récursif Θ tel que $\Theta(\alpha, p, x)(y)\downarrow$ pour toute α et tout p, x, y. D'autre part, l'opérateur récursif Γ satisfaisant $\Gamma(\alpha, \beta, p) = \beta$ est clairement tel que pour tout ψ, p_1, p_2 et q, $\psi_q = \Gamma(\psi(p_1), \psi(q), p_2)$. On a donc

$$\mathcal{R}(\psi, p_1, p_2, q) \Longleftrightarrow \psi_q = \Gamma(\psi(p_1), \psi(q), p_2).$$

L'existence de Γ nous permet de conclure que rs satisfait la Définition 2.33 (a) et est donc extensionnelle avec récursion au sens de Royer. Par le Théorème 2.36, elle est donc de Riccardi. Comme rs n'est clairement pas univaluée, alors, par le Corollaire 2.43, elle n'est pas extensionnelle sans récursion au sens de Royer. □

La partie (a) de la proposition précédente, avec le Théorème 2.37, fournit le corollaire suivant.

2.47 Corollaire *Toute RVE extensionnelle univaluée est extensionnelle sans récursion.*

Voici quelques commentaires informels sur la preuve de la Proposition 2.46 (b). Soient Γ et Θ comme à cet endroit. L'opérateur récursif Γ réussit à admettre n'importe quelle fonction partielle β en "regardant d'avance" β grâce à son argument auto-référenciel. Par contraste, Θ permet explicitement toutes les sorties possibles *de même que la divergence* sur tous les arguments, sans "tricher". Donc, un opérateur récursif témoignant de l'extensionnalité avec récursion au sens de Royer d'une RS peut admettre plusieurs sorties différentes sur certains arguments, en se servant de son paramètre d'auto-référence. Cependant, on peut montrer que l'opérateur récursif ne peut jouer ce jeu sur une entrée *que si la divergence est aussi admissible* pour cette entrée, et c'est essentiellement cette observation qu'établit le prochain théorème. Auparavant, cependant, nous avons besoin de définir une dernière forme d'extensionnalité, qui revient intuitivement à "empêcher" l'opérateur récursif déterminant extensionnellement une RVE, d'admettre à la fois la convergence et la divergence sur une entrée (voir la discussion intuitive suivant la Définition 2.15).

2.48 Définition (a) Soient Γ un opérateur récursif, $\alpha, \beta \in \mathcal{PartRecUn}$ et $p \in N$. On dit que β est *fortement compatible extensionnellement sans récursion avec* (α, p) ssi

$$(\forall x)[[\beta(x)\downarrow \Rightarrow \Gamma(\alpha, p, x)(\beta(x))\downarrow] \\ \& [\beta(x)\uparrow \Rightarrow \neg(\exists y)[\Gamma(\alpha, p, x)(y)\downarrow]]]].$$

(b) On dit qu'un opérateur récursif Γ *détermine fortement extensionnellement sans récursion* une RS (à deux entrées) $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ ssi pour tout ψ, p_1, p_2, q,

$$\mathcal{R}(\psi, p_1, p_2, q) \iff \psi(q) \text{ est fortement compatible extensionnellement sans récursion avec } (\psi(p_1), p_2) \text{ via } \Gamma.$$

(c) Une RS est dite *fortement extensionnelle sans récursion* ssi il existe un opérateur récursif qui la détermine fortement extensionnellement sans récursion.

On vérifie facilement que :

2.49 Observation Toute RS fortement extensionnelle sans récursion est aussi extensionnelle sans récursion, et donc, par l'Observation 2.45, une RVE extensionnelle.

Enfin, le théorème suivant montre que la notion d'extensionnalité forte sans récursion, restreinte aux RS de Riccardi, correspond, elle, à l'extensionnalité sans récursion au sens de Royer.

2.50 Théorème *Une RS est extensionnelle sans récursion au sens de Royer ssi elle est de Riccardi et fortement extensionnelle sans récursion.*

Preuve. $\Rightarrow$: Une modification très simple de la preuve de la Proposition 2.46 (a) permet d'obtenir le résultat voulu.

$\Leftarrow$: Soit $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ une RS de Riccardi déterminée fortement extensionnellement sans récursion par un opérateur récursif Γ. Par l'Observation 2.49, rs est une RVE extensionnelle. Étant de Riccardi et extensionnelle, rs est, par

le Théorème 2.36, extensionnelle avec récursion au sens de Royer. Soit donc Θ un opérateur récursif qui détermine rs extensionnellement avec récursion au sens de Royer. Soit aussi R la relation sur $N \times \mathcal{PartRecUn}^2$ témoignant de l'extensionnalité de rs.

Nous utilisons l'opération sur les ensembles finis de couples $UNIVAL$, définie dans la preuve du Théorème 2.37. Également, l'expression "$\Gamma(\alpha, p, x)(y)\downarrow$ en au plus i étapes" signifie qu'une règle "D_u cause $\langle p, x, y, z\rangle$" pour un certain z et un certain $D_u \subseteq \alpha$ apparaît dans les i premières étapes d'une énumération standard de W_a, où Φ_a est l'opérateur d'énumération déterminant Γ.

Nous montrons d'abord que pour toute $\alpha \in \mathcal{PartRecUn}$ et tout p, x, il existe au plus un y pour lequel $\Gamma(\alpha, p, x)(y)\downarrow$. Supposons le contraire, i.e., qu'il existe $\alpha \in \mathcal{PartRecUn}$, p, x_0, y_0 et y_1 tels que $y_0 \neq y_1$, $\Gamma(\alpha, p, x_0)(y_0)\downarrow$ et $\Gamma(\alpha, p, x_0)(y_1)\downarrow$. Par continuité des opérateurs récursifs, on peut choisir α finie. Pour chaque γ finie et chaque i, définissons

$$E(\gamma, i) \stackrel{\mathrm{d}}{=} \{(x, y) | x \notin \mathbf{dom}(\gamma) \ \& \ \Gamma(\alpha, p, x)(y)\downarrow \text{ en au plus } i \text{ étapes}\}.$$

Pour chaque $j \leq 1$ et chaque i, on définit récursivement $F(j, i)$:

$$\begin{array}{lcl} F(j, 0) & \stackrel{\mathrm{d}}{=} & \{(x_0, y_j)\}, \\ F(j, i+1) & \stackrel{\mathrm{d}}{=} & F(j, i) \cup UNIVAL(E(F(j, i), i)). \end{array}$$

On définit finalement, pour chaque $j \leq 1$, $\beta_j \stackrel{\mathrm{d}}{=} \bigcup_{i \geq 0} F(j, i)$. De façon similaire à dans la preuve du Théorème 2.37, on peut montrer que :

2.51 Lemme *Pour chaque $j \leq 1$,*

(a) $(x_0, y_j) \in \beta_j$.
(b) β_j *est partielle récursive.*
(c) β_j *est fortement compatible extensionnellement avec* (α, p) *via* Γ.

D'autre part, par des techniques standard, on exhibe facilement un opérateur récursif Ψ tel que pour toute β, $\Psi(\beta) = \Theta(\alpha, \beta, p)$. Par [Rog87, Théorème 11.XII], Ψ possède un *point fixe minimal partiel récursif*, i.e., une fonction partielle récursive δ telle que $\Psi(\delta) = \delta$, et pour toute δ' satisfaisant $\Psi(\delta') = \delta'$, $\delta \subseteq \delta'$.

Fixons $\beta \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$. On observe que β est un point fixe de Ψ ssi $\Theta(\alpha, \beta, p) = \beta$. D'autre part, comme Θ détermine rs extensionnellement avec récursion au sens de Royer, $\Theta(\alpha, \beta, p) = \beta$ ssi $R(p, \alpha, \beta)$. Donc, β est un point fixe de Ψ ssi $R(p, \alpha, \beta)$. On déduit qu'en particulier, $R(p, \alpha, \delta)$.

Supposons d'abord que $x_0 \notin \mathbf{dom}(\delta)$. Comme $R(p, \alpha, \delta)$ et puisque Γ détermine rs fortement extensionnellement sans récursion, δ est fortement compatible sans récursion avec (α, p). Alors, par la Définition 2.48, $x_0 \notin \mathbf{dom}(\delta)$ implique $\neg(\exists y)[\Gamma(\alpha, p, x_0)(y)\downarrow]$, ce qui contredit le fait que $\Gamma(\alpha, p, x_0)(y_0)\downarrow$. Donc $x_0 \in \mathbf{dom}(\delta)$.

Clairement, il existe $j \leq 1$ tel que $y_j \neq \delta(x_0)$. Par le Lemme 2.51 (b) et (c), β_j est partielle récursive et fortement compatible sans récursion avec (α, p). Donc, par le fait que Γ détermine rs fortement extensionnellement sans récursion, $R(p, \alpha, \beta_j)$; ce qui entraîne, par l'observation faite précédemment, que β_j est un point fixe de Ψ. Donc, $\delta \subseteq \beta_j$. D'autre part, par le Lemme 2.51 (a), $(x_0, y_j) \in \beta_j$. Donc, $\{(x_0, \delta(x_0)), (x_0, y_j)\} \subseteq \beta_j$ et $y_j \neq \delta(x_0)$: une contradiction.

Nous concluons donc que pour toute $\alpha \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$ et tout p, x, il existe au plus un y pour lequel $\Gamma(\alpha, p, x)(y)\downarrow$. Par une variante immédiate de la Proposition 2.16, on déduit que pour toute α, β et tout p,

$$R(p, \alpha, \beta) \iff (\forall x, y)[\Gamma(\alpha, p, x)(y)\downarrow \iff y = \beta(x)]. \tag{2.9}$$

Par des techniques standard, on exhibe facilement un opérateur récursif Φ tel que pour toute α, x et y,

$$\Phi(\alpha, p)(x) = y \iff \Gamma(\alpha, p, x)(y)\downarrow. \tag{2.10}$$

Par (2.9) et (2.10), on déduit que Φ détermine rs extensionnellement sans récursion au sens de Royer, et que donc, rs est extensionnelle sans récursion au sens de Royer. □ **Théorème 2.50**

2.52 Corollaire *Une RS fortement extensionnelle sans récursion et non-univaluée ne coïncide quant aux instances avec aucun schéma de structure de contrôle (et n'est donc pas de Riccardi).*

Preuve. Nous montrons que toute RS fortement extensionnelle sans récursion qui coïncide quant aux instances avec un schéma de structure de contrôle est

extensionnelle sans récursion au sens de Royer. Comme toute telle RS est univaluée (par le Théorème 2.37), le corollaire s'ensuivra.

Soit $\mathsf{rs} = (\mathcal{R}, \mathit{VRAI})$ une RS fortement extensionnelle sans récursion coïncidant quant aux instances avec un schéma de structure de contrôle (Θ, P). On procède comme dans la preuve de la partie "$\Leftarrow$" du théorème, sauf qu'on choisit Ψ comme étant l'opérateur récursif tel que, pour toute $\beta \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$, $\Psi(\beta) = \Theta(\phi[\alpha, \beta], 0, p, 1)$, où α et p sont comme dans la preuve du théorème, et où pour toutes $\zeta, \eta \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$, $\phi[\zeta, \eta]$ dénote le SP dont tous les programmes calculent la même chose qu'en ϕ, sauf 0 et 1, qui calculent respectivement ζ et η. On montre alors comme suit que pour toute $\beta \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$, β est un point fixe de Ψ ssi $R(p, \alpha, \beta)$, où R est comme dans la preuve du théorème.

En se servant des faits que rs et (Θ, P) coïncident quant aux instances et que rs a prédicat VRAI, on montre facilement que pour tout ψ ayant une instance de rs et tous p_1, p_2 et q, $\mathcal{R}(\psi, p_1, p_2, q) \iff \Theta(\psi, p_1, p_2, q) = \psi_q$. Comme $\phi[\alpha, \beta]$ est acceptable (par la Proposition 1.14) et comme tout SP acceptable possède une instance de rs (par le Théorème 2.69), on a donc $\mathcal{R}(\phi[\alpha, \beta], 0, p, 1) \iff \Theta(\phi[\alpha, \beta], 0, p, 1) = \phi[\alpha, \beta]_1$. Comme R témoigne de l'extensionnalité de rs, comme $\phi[\alpha, \beta]_1 = \beta$, et par notre choix de Ψ, on obtient $R(p, \alpha, \beta) \iff \Psi(\beta) = \beta$.

Poursuivant comme dans la preuve de la partie "$\Leftarrow$" du théorème, on arrive à la conclusion que toute RS fortement extensionnelle sans récursion coïncidant quant aux instances avec un schéma de structure de contrôle est extensionnelle sans récursion au sens de Royer. $\square$

Nous montrons maintenant que $\mathsf{ps\text{-}inv}$ est fortement extensionnelle sans récursion. Par l'Observation 2.49, ceci établira que $\mathsf{ps\text{-}inv}$ est une RVE. Comme clairement elle n'est pas univaluée, puisqu'une fonction partielle récursive a en général plus d'une pseudo-inverse partielle récursive, ceci établira aussi, par le Corollaire 2.52, qu'elle n'est pas exprimable comme structure de contrôle.

2.53 Théorème $\mathsf{ps\text{-}inv}$ *est une RS purement et fortement extensionnelle sans récursion.*

Preuve. Par les définitions pertinentes, il est suffisant de montrer l'existence

d'un opérateur récursif Θ tel que pour toute α et β,

$$\begin{aligned}&[\beta \text{ est une pseudo-inverse de } \alpha] \Longleftrightarrow \\ &\quad (\forall x)[[\beta(x)\downarrow \Rightarrow \Theta(\alpha,x)(\beta(x))\downarrow] \\ &\qquad\qquad \& \ [\beta(x)\uparrow \Rightarrow \neg(\exists y)[\Theta(\alpha,x)(y)\downarrow]]].\end{aligned} \tag{2.11}$$

Soit Θ l'opérateur récursif déterminé par l'opérateur d'énumération Φ_z, où W_z contient précisément et seulement les règles "$\langle y,x\rangle$ cause $\langle x,y,0\rangle$" pour tout x et y. Nous montrons que Θ satisfait (2.11).

$\Rightarrow$: Supposons que β est une pseudo-inverse de α et fixons x. Si $\beta(x)\downarrow$, alors il existe y tel que $\beta(x) = y$, et alors, par définition de pseudo-inverse (Définition 2.6), $\alpha(y) = x$, i.e., $(y,x) \in \alpha$. Donc, par la règle "$\langle y,x\rangle$ cause $\langle x,y,0\rangle$" dans W_z, on aura $\Theta(\alpha,x)(y)\downarrow$. D'autre part, si $\beta(x)\uparrow$, alors il n'existe aucun y tel que $\beta(x) = y$, et alors, par définition de pseudo-inverse, il n'existe aucun y tel que $(y,x) \in \alpha$. Donc, par définition de Θ, il n'existe aucun y tel que $\Theta(\alpha,x)(y)\downarrow$.

$\Leftarrow$: Supposons que α et β satisfont

$$(\forall x)[[\beta(x)\downarrow \Rightarrow \Theta(\alpha,x)(\beta(x))\downarrow] \ \& \ [\beta(x)\uparrow \Rightarrow \neg(\exists y)[\Theta(\alpha,x)(y)\downarrow]]], \tag{2.12}$$

et fixons x. Si $\beta(x)\downarrow$, alors par (2.12) et la définition de Θ, $\alpha(\beta(x)) = x$. Donc, $\mathbf{dom}(\beta) \subseteq \mathbf{image}(\alpha)$ et pour tout $x \in \mathbf{dom}(\beta)$, $\alpha(\beta(x)) = x$. Si $\beta(x)\uparrow$, alors par (2.12) et la définition de Θ, il n'existe aucun y tel que $\alpha(y) = x$, c'est-à-dire que $x \notin \mathbf{image}(\alpha)$. Donc, $\mathbf{image}(\alpha) \subseteq \mathbf{dom}(\beta)$, d'où $\mathbf{image}(\alpha) = \mathbf{dom}(\beta)$ et, par la définition de pseudo-inverse, β est une pseudo-inverse de α. □

Les RS rech-nb et majoration sont toutes deux fortement extensionnelles sans récursion, et donc, par l'Observation 2.49, des RVE. Les preuves de ces faits sont très similaires à celle du Théorème 2.53, et sont omises. Tout comme pour ps-inv, les faits que rech-nb et majoration ne sont pas exprimables comme structures de contrôle découlent, via le Corollaire 2.52, des faits (immédiats) qu'elles ne sont pas univaluées.

Les RS comp, s-m-1 et prog sont extensionnelles sans récursion au sens de Royer ; remb, remb-inf et acc ne sont bien sûr pas extensionnelles, puisque leur prédicat n'est pas *VRAI* ; Royer a montré (essentiellement) que krt et n-pkrt ne sont pas extensionnelles [Roy87, § 2.3].

Nous terminons cette section en définissant une classe de RS que l'on pourrait qualifier des plus "élémentaires" qui soient.

2.54 Définition Une RS est dite *de Myhill-Shepherdson* ssi elle est à une entrée et purement extensionnelle sans récursion au sens de Royer.

Les RS de Myhill-Shepherdson correspondent exactement à la même notion de transformation sémantique que les opérateurs récursifs "nature", i.e., non généralisés par la Notation 1.16.

Nous utilisons l'appellation *de Myhill-Shepherdson* parce que Myhill et Shepherdson ont obtenu dès 1955 un résultat impliquant que toutes les RS à une entrée purement extensionnelles sans récursion au sens de Royer sont récursivement satisfaisables dans un SP acceptable spécifique [MS55, Rog87, p. 196]. Leur preuve se généralise facilement à tout SP acceptable, et donne donc un "petit" théorème de complétude expressive.

2.6 Relations de garantie

Les définitions et la notation suivantes sont, avec quelques variations, les transpositions, dans le monde des RS, de définitions et de notation de Royer [Roy87, § 1.3].

2.55 Définition Pour tout ψ, $\mathsf{UTIL}(\psi) \stackrel{\mathrm{d}}{=} \{\mathsf{rs} \mid \mathsf{rs}$ est récursivement satisfaisable en $\psi\}$.

Intuitivement, $\mathsf{UTIL}(\psi)$ représente les techniques de programmation utilisables dans le SP ψ.

2.56 Définition Soient A et B deux ensembles de RS et $\mathcal{S}$ un ensemble de SP.

(a) On dit que A *garantit* B *dans* $\mathcal{S}$, noté $\mathsf{A} \models_{\mathcal{S}} \mathsf{B}$, ssi pour tout $\psi \in \mathcal{S}$, $\mathsf{A} \subseteq \mathsf{UTIL}(\psi) \Longrightarrow \mathsf{B} \subseteq \mathsf{UTIL}(\psi)$.

(b) On dit que A *garantit une de* B *dans* $\mathcal{S}$, noté $\mathsf{A} \models\!\equiv_{\mathcal{S}} \mathsf{B}$, ssi pour tout $\psi \in \mathcal{S}$, $\mathsf{A} \subseteq \mathsf{UTIL}(\psi) \Longrightarrow \mathsf{B} \cap \mathsf{UTIL}(\psi) \neq \emptyset$.

Supposons que A est l'ensemble de tous les ensembles de RS. Pour chaque $\mathcal{S} \subseteq \mathcal{SP}$, on peut voir $\models_{\mathcal{S}}$ et $\models\!\equiv_{\mathcal{S}}$ comme des relations sur A^2. En tant

que relation, $\models_{\mathcal{S}}$ est une relation d'ordre partiel. On utilise le symbole $\models$ (respectivement, $\equiv$) pour parler d'une relation $\models_{\mathcal{S}}$ (respectivement, $\equiv_{\mathcal{S}}$) pour un $\mathcal{S} \subseteq \mathcal{SP}$ quelconque, sans le spécifier.

Un symbole dénotant une RS utilisé à gauche ou à droite de $\models$, ou à gauche de $\equiv$ représente le singleton contenant cette RS. Ainsi, par exemple, $\textsf{rs-a} \models \textsf{B}$ signifie $\{\textsf{rs-a}\} \models \textsf{B}$.

Les symboles $\not\models$ et $\not\equiv$ représentent les compléments de respectivement $\models$ et $\equiv$; par exemple, pour tout $\textsf{A}$ et $\textsf{B}$, $\textsf{A} \not\models \textsf{B} \iff \neg(\textsf{A} \models \textsf{B})$.

Comme nous parlerons beaucoup d'interrelations de garantie dans les SP exécutables ou maximaux ou les deux, nous introduisons la notation suivante :

2.57 Notation Les symboles $\models_e$, $\models_m$, $\models_{em}$, $\equiv_m$ dénotent respectivement les relations $\models_{\mathcal{SPE}}$, $\models_{\mathcal{SPM}}$, $\models_{\mathcal{SPEM}}$, $\equiv_{\mathcal{SPM}}$.

Pour tout $\textsf{A}$ et $\textsf{B}$, on dit que $\textsf{A}$ *garantit* $\textsf{B}$ *dans les SP exécutables (respectivement, maximaux, exécutables maximaux)* ssi $\textsf{A} \models_e \textsf{B}$ (respectivement, $\textsf{A} \models_m \textsf{B}$, $\textsf{A} \models_{em} \textsf{B}$).

On dit que $\textsf{rs-a}$ et $\textsf{rs-b}$ sont *indépendantes* ssi $\textsf{rs-a} \not\models \textsf{rs-b}$ et $\textsf{rs-b} \not\models \textsf{rs-a}$.

La *puissance expressive* d'une RS $\textsf{rs}$ est l'ensemble de toutes les RS garanties par $\textsf{rs}$. On dira qu'une RS $\textsf{rs-a}$ est plus *forte* (respectivement, plus *faible*) qu'une $\textsf{rs-b}$ ssi sa puissance expressive est plus grande (respectivement, plus petite).

Si on dit qu'une certaine propriété des SP "garantit" une RS $\textsf{rs}$, on veut dire que le fait qu'un SP soit doté de cette propriété implique qu'il a une instance effective de $\textsf{rs}$. Inversement, on dit qu'une RS $\textsf{rs}$ "garantit" une propriété des SP ssi le fait qu'un SP ait une instance effective de $\textsf{rs}$ implique qu'il est doté de la propriété en question.

2.7 Similitude et versatilité des SP acceptables

Nous avons en partie justifié l'importance de la pseudo-inversion par le fait que c'est une technique utilisable dans n'importe quel SP acceptable. Ceci motive la définition suivante :

2.58 Définition $\mathsf{PORT} \stackrel{\mathrm{d}}{=} \{\mathsf{rs} \mid \text{pour tout } \psi \in \mathcal{SPA}, \mathsf{rs} \in \mathsf{UTIL}(\psi)\}$.

Intuitivement, PORT correspond à la classe des techniques de programmation "portables", i.e., utilisables dans n'importe langage de programmation acceptable. Tel que discuté dans l'introduction, "l'étendue" de PORT nous renseigne à la fois sur la similitude des SP acceptables entre eux et sur leur versatilité, i.e., la variété des techniques de programmation qui y sont utilisables.

La plupart des résultats sur l'étendue de PORT donnent lieu à une *caractérisation* de l'acceptabilité, comme suit : on montre, pour un "gros" ensemble A de RS, que $\mathsf{A} \subseteq \mathsf{PORT}$. Mais il s'avère (puisque A est "gros") que pour un certain sous-ensemble de A, mettons B, $\mathsf{B} \models_e \mathsf{acc}$. On conclut donc qu'un SP ψ est acceptable ssi il est exécutable et $\mathsf{A} \subseteq \mathsf{UTIL}(\psi)$.

Par exemple, définissons :

2.59 Définition MR-RIC dénote l'ensemble des RS de Riccardi à prédicat MR.

Clairement, $\mathsf{prog} \in \mathsf{MR\text{-}RIC}$, et clairement $\mathsf{prog} \models_e \mathsf{acc}$. On tire donc du Théorème 2.5 la caractérisation suivante des SP acceptables : un SP ψ est acceptable ssi il est exécutable et $\mathsf{MR\text{-}RIC} \subseteq \mathsf{UTIL}(\psi)$ [Roy87, Théorème 1.4.3.16]. L'observation que nous allons maintenant faire permet de renforcer légèrement ces caractérisations en éliminant la référence à l'exécutabilité.

Machtey et Young [MY78] définissent un SP "universel" comme étant un SP maximal possédant un "programme universel", i.e., un programme calculant la fonction universelle (unaire) du SP. Les auteurs définissent ensuite un SP acceptable comme étant un SP universel qui est programmable, et montrent

que leur définition est équivalente à la définition de Rogers [Rog58]. Il n'a apparemment jamais été remarqué que la possession d'un programme universel peut être exprimée comme une RS de Riccardi (donc aussi, comme une structure de contrôle).

2.60 Définition $\mathsf{univ} \stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p, q, $\mathcal{R}(\psi, p, q) \stackrel{\mathrm{d}}{\Longleftrightarrow} \psi_q = \widehat{\psi}$.

2.61 Proposition univ *est de Riccardi ; en fait,* $\mathsf{univ} \in \mathsf{MR\text{-}RIC}$.

Preuve. L'opérateur récursif Θ tel que pour tout α, p, q, $\Theta(\alpha, p, q) = \alpha$ détermine univ à la Riccardi. Comme univ a prédicat VRAI, qui est clairement MR, on conclut que $\mathsf{univ} \in \mathsf{MR\text{-}RIC}$. □

Intuitivement, $\mathsf{univ} \in \mathsf{UTIL}(\psi)$ ssi on peut "écrire un programme universel" en ψ. Il faut bien noter la nuance entre cette interprétation intuitive, et celle (invalide) qui dirait que $\mathsf{univ} \in \mathsf{UTIL}(\psi)$ ssi ψ est exécutable. En effet, le SP dans lequel tout programme p calcule la fonction constante $\lambda z.p$, bien qu'exécutable, ne possède aucune instance de univ. Bien que très rapprochée de l'exécutabilité, univ représente donc bel et bien une propriété de *programmation* des SP.

Le fait que la définition de Machtey et Young est équivalente à celle de Rogers nous donne :

2.62 Proposition *Pour tout* ψ, $\{\mathsf{univ}, \mathsf{prog}\} \subseteq \mathsf{UTIL}(\psi) \Longleftrightarrow \psi \in \mathcal{SPA}$.

Comme prog est clairement aussi dans $\mathsf{MR\text{-}RIC}$, le Théorème 2.5 donne donc :

2.63 Corollaire *Un SP* ψ *est acceptable ssi* $\mathsf{MR\text{-}RIC} \subseteq \mathsf{UTIL}(\psi)$.

Ce résultat est généralisé par notre Corollaire 2.70 ci-dessous.

Un autre genre d'interrelations de garantie impliquant acc sont celles où acc se retrouve à *droite* du symbole $\models$. Intuitivement, ces résultats nous renseignent sur ce qu'il faut mettre au minimum dans un langage pour qu'il soit acceptable.

Il est bien connu que certains "petits" ensembles de RS (de Riccardi) garantissent l'acceptabilité. En fait, la Proposition 2.62 donne un tel ensemble. Cependant, alors que la garantie fournie par la Proposition 2.62 est valide pour *tous* les SP, la plupart des autres garanties à "saveur minimale" de l'acceptabilité sont démontrées dans le contexte des SP exécutables maximaux. Il est bien connu que comp et s-1-1 garantissent l'acceptabilité dans ce contexte. Royer a montré qu'il existe une RS *de Myhill-Shepherdson* (donc, intuitivement, très simple) qui, *à elle seule*, garantit l'acceptabilité dans les SP exécutables maximaux [Roy87, Théorème 4.2.2 (b)] (ni comp ni s-1-1 ne sont de Myhill-Shepherdson). Cependant, nous verrons au Chapitre 4 qu'aucune garantie de prog par une unique RS de Myhill-Shepherdson ne peut être "robuste", i.e., être valide dans le contexte des SP maximaux qui ne sont pas nécessairement exécutables (Théorème 4.12).

Nous démontrons maintenant que pour une vaste classe de RVE, toutes sont "portables", i.e., utilisables dans n'importe quel SP acceptable. Ce résultat améliore par deux aspects le meilleur résultat du genre connu précédemment, le Théorème 1.4.3.16 de Royer dans [Roy87]. D'une part il s'applique à certaines RS qui ne sont pas de Riccardi alors que le résultat de Royer ne s'applique qu'à des RS de Riccardi, d'autre part, il englobe une classe de prédicats strictement plus grande que celle des prédicats MR (Définition 2.4), visée par le résultat de Royer.

Nous commençons pas définir une classe de prédicats sur les fonctions, et démontrer qu'elle inclut strictement celle des prédicats MR. Nous avons besoin de quelques définitions accessoires.

2.64 Définition Un prédicat Q sur les séquences est dit *cumulativement co-fini* ssi il est récursif, $Q(\nu)$ est vrai, et pour toute σ, $Q(\sigma) \Longrightarrow (\exists x_0)(\forall x \geq x_0)[Q(\sigma : x)]$.

2.65 Définition Soit $m > 0$. Une fonction τ est une *fonction de pairage m-aire* ssi elle est une bijection récursive de N^m vers N.

2.66 Définition Soit $m > 0$. Un prédicat P sur les fonctions m-aires est dit *prédicat CC* ssi il existe un prédicat sur les séquences cumulativement co-fini Q et une fonction de pairage m-aire τ tels que pour toute fonction m-aire f,

$$[(\forall x)Q(f(\tau^{-1}(0)) : f(\tau^{-1}(1)) : \cdots : f(\tau^{-1}(x)))] \Longrightarrow P(f).$$

Voici une interprétation intuitive du fait qu'un prédicat P sur les fonctions m-aires soit CC : on a d'abord une façon fixe de balayer l'espace N^m (la fonction de pairage m-aire). Pour savoir si une fonction m-aire f satisfait P, on balaie N^m en regardant à chaque point rencontré la valeur de f à ce point. Si à chaque point, la séquence des valeurs rencontrées jusque là satisfait une certaine condition de validité, alors $P(f)$ est vrai. La "certaine condition de validité" est telle que presque toutes les prolongations (par une valeur) d'une séquence valide sont valides.

2.67 Proposition *Tout prédicat MR est un prédicat CC.*

Preuve. Soit $m > 0$. Soit P le prédicat sur les fonctions m-aires tel que pour toute f, $P(f)$ ssi f est injective et strictement croissante en chaque argument. Il suffit de montrer que P est un prédicat CC pour établir la proposition. Définissons $\tau \stackrel{\mathrm{d}}{=} \lambda x_1, \ldots, x_m.\langle x_1, \ldots, x_m\rangle$. Il s'agit clairement d'une fonction de pairage m-aire. Définissons le prédicat sur les séquences Q comme suit :

$$\begin{array}{lcll} Q(\nu) & \stackrel{\mathrm{d}}{\Longleftrightarrow} & \text{vrai}, & \\ Q(\nu : x) & \stackrel{\mathrm{d}}{\Longleftrightarrow} & \text{vrai}, & \text{pour tout } x, \\ Q(\sigma : x) & \stackrel{\mathrm{d}}{\Longleftrightarrow} & x > \sigma_{\ell(\sigma)}, & \text{pour tout } x \text{ et } \sigma \neq \nu. \end{array}$$

Par un raisonnement de routine, on vérifie que Q est cumulativement co-fini et que pour tout f m-aire,

$$[(\forall x)Q(f(\tau^{-1}(0)) : f(\tau^{-1}(1)) : \cdots : f(\tau^{-1}(x)))] \Longrightarrow P(f).$$

Ceci nous permet de conclure que P est un prédicat CC. □

Tout prédicat CC n'est cependant pas MR. En effet, il est facile de vérifier que le prédicat sur les fonctions unaires $\lambda f.[(\forall x)[f(x) \neq x]]$ est CC mais *pas* MR, puisque la fonction identité est injective et strictement croissante, mais ne satisfait pas ce prédicat.[4] Le prédicat $\lambda f.[(\forall n)[f(2n) > f(2n+1)]]$ et celui de l'Exemple 2.2 (le schéma de structure de contrôle $COMP$) sont d'autres exemples de prédicat CC mais non MR (pour démontrer que $\lambda f.[(\forall n)[f(2n) > f(2n+1)]]$ est CC, il faut utiliser une fonction de pairage

4. Royer mentionne par erreur que le prédicat $\lambda f.[(\forall x)[f(x) \neq x]]$ est un prédicat MR [Roy87, p. 24].

différente de l'identité). L'ensemble des prédicats CC contient donc proprement celui des prédicats MR.

Le prédicat $\lambda f.[(\forall x)[f(x) \neq x]]$ est celui de remb. Il est facile de montrer que les prédicats de $\mathsf{remb\text{-}inf}$ et $\mathsf{remb\text{-}inf\text{-}inj}$ sont aussi CC mais non MR. Notre motivation première pour l'introduction des prédicats CC est l'inclusion des RS remb, $\mathsf{remb\text{-}inf}$ et $\mathsf{remb\text{-}inf\text{-}inj}$ dans un théorème de complétude expressive. Mais nous avons aussi, indépendamment de ces RS spécifiques, essayé de définir la notion la plus générale de prédicat qui puisse tomber sous le coup d'un théorème de complétude expressive.

Voici ce théorème, qui nous renseigne sur l'étendue de la similitude et de la versatilité d'utilisation des SP acceptables.

2.68 Définition $\mathsf{CC\text{-}RVE} \stackrel{\mathrm{d}}{=} \{(\mathcal{R}, P) \mid (\mathcal{R}, P) \in \mathsf{RVE}$ et P est un prédicat CC$\}$.

2.69 Théorème $\mathsf{CC\text{-}RVE} \subseteq \mathsf{PORT}$.

Preuve. Soient $\mathsf{rs} = (\mathcal{R}, P) \in \mathsf{CC\text{-}RVE}$ et $\psi \in \mathcal{SPA}$. Supposons que rs est à m entrées. Soient Θ un opérateur récursif déterminant rs, et τ et Q respectivement une fonction de pairage m-aire et un prédicat sur les séquences cumulativement co-fini tels que pour toute f m-aire,

$$[(\forall x)Q(f(\tau^{-1}(0)) : f(\tau^{-1}(1)) : \cdots : f(\tau^{-1}(x)))] \Longrightarrow P(f). \tag{2.13}$$

Royer a montré que $\mathsf{1\text{-}pkrt\text{-}inj} \in \mathsf{PORT}$. Donc, $\mathsf{1\text{-}pkrt\text{-}inj} \in \mathsf{UTIL}(\psi)$. Soit r une instance effective de $\mathsf{1\text{-}pkrt\text{-}inj}$ dans ψ. Comme ψ est exécutable et de puissance de calcul maximale, il existe un ψ-programme d réalisant l'algorithme suivant :

> **Entrée :** $\langle q, \langle p_1, \ldots, p_m, n\rangle, x\rangle$
> **Algorithme pour** ψ_d **:** Par queue de colombe, calculer $\Theta(\psi, p_1, \ldots, p_m, q, x)(y + 1)$ pour chaque $y \in N$. Dès qu'un résultat est obtenu pour un certain $y_0 + 1$, retourner ce y_0.
> $\square$ **Algorithme**

Soit maintenant f la fonction calculée par l'algorithme (récursif) suivant :

Entrée : $(p_1, \ldots, p_m)$
Algorithme pour f **:**

1. Si $\tau(p_1, \ldots, p_m) = 0$, alors $\sigma \leftarrow \nu$ et passer à l'Étape 4.
2. $z \leftarrow \tau(p_1, \ldots, p_m)$.
3. $\sigma \leftarrow f(\tau^{-1}(0)) : \cdots : f(\tau^{-1}(z-1))$.
4. $n \leftarrow (\mu k)[Q(\sigma : r(d, \langle p_1, \ldots, p_m, k\rangle))]$.
5. Retourner $r(d, \langle p_1, \ldots, p_m, n\rangle)$.

□ **Algorithme**

Il est clair que la récursion dans l'algorithme n'entraîne aucune circularité. Comme r est injective, comme Q est cumulativement co-fini, et comme les deux sont récursifs, l'Étape 4 se termine toujours. Il s'ensuit donc que f est récursive. D'autre part, par l'algorithme pour f, il est vrai que pour tout x, $Q(f(\tau^{-1}(0)) : f(\tau^{-1}(1)) : \cdots : f(\tau^{-1}(x)))$ et donc, par (2.13), $P(f)$.

D'autre part, pour tout $p_1, \ldots, p_m$, $f(p_1, \ldots, p_m)$ est de la forme $r(d, \langle p_1, \ldots, p_m, n\rangle)$ pour un certain n ; alors, par le fait que r est une instance de 1-pkrt-inj dans ψ et par la sémantique du ψ-programme d, on a que pour tout $p_1, \ldots, p_m$, le ψ-programme $f(p_1, \ldots, p_m)$ calcule la fonction partielle correspondant à l'algorithme suivant :

Entrée : x
Algorithme pour $\psi(f(p_1, \ldots, p_m))$ **:**
Par queue de colombe, calculer $\Theta(\psi, p_1, \ldots, p_m, f(p_1, \ldots, p_m), x)(y+1)$ pour tout $y \in N$. Dès qu'un résultat est obtenu pour un certain $y_0 + 1$, retourner ce y_0. □ **Algorithme**

Par la Proposition 2.17, on déduit que f est une instance effective de rs dans ψ. □ **Théorème 2.69**

Comme $\{\mathsf{univ}, \mathsf{prog}\} \subseteq$ MR-RIC $\subseteq$ CC-RVE, nous déduisons la caractérisation suivante de l'acceptabilité :

2.70 Corollaire *Pour tout SP* ψ, $\psi \in \mathcal{SPA} \iff$ CC-RVE $\subseteq$ UTIL(ψ).

Nous avons encore :

2.71 Corollaire prog $\models_e$ CC-RVE.

Preuve. Tout SP exécutable programmable est par définition acceptable. Donc, pour tout $\psi \in \mathcal{SPE}$, si $\mathsf{prog} \in \mathsf{UTIL}(\psi)$, alors $\psi \in \mathcal{SPA}$, et donc, par le théorème, $\mathsf{CC\text{-}RVE} \subseteq \mathsf{UTIL}(\psi)$. □

2.8 Discussion

Nous mentionnons que Riccardi a introduit un certain nombre d'extensions à la structure de contrôle, dont une version à plusieurs sorties, c'est-à-dire où les instances retournent des *tuplets* de programmes. La version originelle de Case du Théorème de complétude expressive pour les structures de contrôle [Ric80, p. 52] s'appliquait aussi à cette extension.

Avec la RVE, et plus particulièrement avec la classe CC-RVE, nous avons identifié un sous-ensemble strictement plus grand de PORT que ce qui était connu auparavant. Clairement, puisque la classe des prédicats CC n'inclut pas tous les prédicats, il "reste" quelque-chose dans $\mathsf{PORT} - \mathsf{CC\text{-}RVE}$, mais quoi ? Nous donnons deux exemples de RS *avec prédicat VRAI* qui sont dans $\mathsf{PORT} - \mathsf{RVE}$. Les deux exemples sont, peut-être par nécessité, un peu "pathologiques".

Soit A un sous-ensemble strict et non-vide de N ; K dénote l'ensemble r.é. non-récursif de § 1.3.7.

2.72 Définition (a) $\mathsf{path1} \overset{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p et q,

$$\mathcal{R}(\psi, p, q) \Longleftrightarrow q \in A.$$

(b) $\mathsf{path2} \overset{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p et q,

$$\mathcal{R}(\psi, p, q) \Longleftrightarrow [q \in K \vee \psi_q = \{(0,0)\}].$$

Comme A est non-vide, la fonction constante $\lambda p.a$, où $a \in A$, est une instance effective de path1 dans n'importe quel SP ; donc $\mathsf{path1} \in \mathsf{PORT}$. Un raisonnement similaire nous amène à conclure que $\mathsf{path2} \in \mathsf{PORT}$. On peut montrer par ailleurs que ni path1 ni path2 ne sont des RVE.

Notons cependant une différence fondamentale entre path1 et path2. Soit $\mathcal{R}$ comme dans la définition de path2. Définissons $\mathcal{R}'$ par $\mathcal{R}'(\psi, p, q) \Longleftrightarrow \psi_q =$

$\{(0,0)\}$, alors clairement, $\mathcal{R}'(\psi, p, q) \Longrightarrow \mathcal{R}(\psi, p, q)$ pour tout ψ, p et q, et donc, toute instance de $(\mathcal{R}', \mathit{VRAI})$ est *aussi* une instance de path2 dans n'importe quel SP. Or, $(\mathcal{R}', \mathit{VRAI}) \in$ CC-RVE. Donc, path2 est en quelque sorte "incluse" dans une RVE de CC-RVE. D'autre part, on peut montrer pour certaines valeurs de A que path1 n'est ainsi "incluse" dans aucune RVE de CC-RVE (par exemple, si A est fini). Il y a donc fondamentalement deux "types" de RS dans PORT − CC-RVE.

Une *classe de Rogers* [Roy87, Définition 1.4.2.1] est une classe de la relation d'équivalence induite sur $\mathcal{SP}$ par la relation d'ordre partiel de traductibilité ($\psi \leq_R \psi' \overset{\mathrm{d}}{\Longleftrightarrow} (\exists f \text{ réc.})(\forall p)[\psi_{f(p)} = \psi'_p]$). Deux membres d'une telle classe sont par définition compilables l'un vers l'autre dans les deux directions. Royer a étudié quelque peu la "transmission" des structures de contrôle d'un membre à l'autre d'une classe de Rogers [Roy87, § 2.3]. Entre autres, il a montré que les structures de contrôle extensionnelles (avec ou sans récursion) sont ainsi "transmises" à l'intérieur d'une classe de Rogers. Il serait certes intéressant de faire la même étude pour les RVE ; c'est un domaine que nous n'avons pas exploré.

Chapitre 3

Complexité des instances : s-1-1 contre composition

3.1 Introduction

Dans ce chapitre, nous nous intéressons à l'aspect *complexité des instances* des interrelations de garantie. Autrement dit, si rs-a $\models$ rs-b, nous nous demandons quelle doit/peut être la complexité d'une instance de rs-b par rapport à celle d'une instance de rs-a dans un même SP. Nous nous intéressons particulièrement aux interrelations s-1-1 $\models_{em}$ CC-RVE et comp $\models_m$ s-1-1.

3.2 Les interrelations s-1-1 $\models_{em}$ CC-RVE

3.2.1 Quelques résultats positifs

Nous commençons par un résultat positif pour une sous-classe de CC-RVE.

3.1 Définition Soit $m > 0$.

(a) Un prédicat $(m+1)$-aire Q est dit *prédicat linéairement co-fini* ssi il est décidable en temps linéaire et il existe un k tel que pour

tout $(p_1, \ldots, p_m)$, il y a au plus k valeurs de x pour lesquelles $\neg Q(p_1, \ldots, p_m, x)$.

(b) Un prédicat sur les fonctions m-aires P est dit *prédicat LC* ssi il existe un prédicat $(m+1)$-aire linéairement co-fini Q tel que pour toute fonction m-aire f,

$$[f \text{ injective } \& \ (\forall p_1, \ldots, p_m) Q(p_1, \ldots, p_m, f(p_1, \ldots, p_m))] \Rightarrow P(f).$$

(c) LC-RVE $\stackrel{\mathrm{d}}{=} \{(\mathcal{R}, P) \mid (\mathcal{R}, P) \in$ RVE et P est un prédicat LC$\}$.

(d) VRAI-RIC $\stackrel{\mathrm{d}}{=} \{(\mathcal{R}, \mathit{VRAI}) \mid (\mathcal{R}, \mathit{VRAI}) \in$ RIC$\}$.

Il est facile de montrer que tout prédicat LC est un prédicat CC. En revanche, on vérifie aisément que le prédicat sur les fonctions unaires $\lambda f.[f$ est strictement croissante$]$ est MR (et donc, CC) mais *pas* LC. Le prédicat de remb, $\lambda f.[(\forall x)[f(x) \neq x]]$, est un exemple de prédicat LC qui n'est *pas* MR. Les prédicats de remb-inf et remb-inf-inj et celui de l'Exemple 2.2 (le schéma de structure de contrôle $COMP$) sont dans le même cas. Clairement, VRAI est un prédicat LC, et donc, LC-RVE contient toutes les RVE à prédicat VRAI, ce qui inclut strictement VRAI-RIC. La classe LC-RVE est donc très riche, et inclut entre autres toutes les RS mentionnées en § 2.4, à l'exception de acc.

Comme pour la classe CC-RVE, notre motivation première pour l'introduction de LC-RVE est d'englober des RS de type "rembourrage". Cependant, indépendamment de ces RS spécifiques, nous avons essayé d'identifier la classe la plus vaste de RS qui pouvaient tomber sous le coup d'une interrelation *efficace* commune de garantie par une RS unique.

Il s'avère que s-1-1 a une puissance expressive exceptionnelle, même en tenant compte de la complexité des instances des RS garanties. Royer a déjà étudié cette puissance expressive dans le contexte des structures de contrôle. Pour comparaison, nous reproduisons ci-dessous quelques conséquences de ses résultats.

3.2 Définition (Royer [Roy87]) mono-s-1-1 $\stackrel{\mathrm{d}}{=} (\mathcal{R}, P)$, où $\mathcal{R}$ est comme dans la définition de s-1-1 (Définition 2.25) et où pour toute f, $P(f) \Longleftrightarrow f$ est injective et strictement croissante en chaque argument.

La RS mono-s-1-1 est intuitivement plus "exigeante" que s-1-1, et sa puissance expressive est *a priori* plus grande que celle de s-1-1.

3.3 Faits (Royer [Roy87]) *Soient $\psi \in \mathcal{SPEM}$, et s et t des instances effectives de respectivement* s-1-1 *et* mono-s-1-1 *dans ψ.*

(a) *Si $s \in \mathcal{L}intime$ (respectivement, $\mathcal{P}time$), alors pour toute* rs $\in$ VRAI-RIC*, ψ a une instance dans $\mathcal{L}intime$ (respectivement, $\mathcal{P}time$) de* rs*.*

(b) *Si $t \in \mathcal{L}intime$ (respectivement, $\mathcal{P}time$), alors pour toute* rs $\in$ MR-RIC*, ψ a une instance dans $\mathcal{L}intime$ (respectivement, $\mathcal{P}time$) de* rs*.*

(c) *Si $s \in \mathcal{E}xptime$ (respectivement, $t \in \mathcal{E}xptime$), alors pour toute* rs $\in$ VRAI-RIC *(respectivement,* MR-RIC*), ψ a une instance dans $\mathcal{E}lm\mathcal{R}ec$ de* rs*.*

(d) *Si $s \in \mathcal{E}lm\mathcal{R}ec$, alors ψ a une instance dans $\mathcal{E}lm\mathcal{R}ec$ de* remb-inf*.*

Nos résultats sur la puissance expressive de s-1-1 s'énoncent plus naturellement pour le cas d'une instance dans $\mathcal{L}intime$. Nous donnons des cas plus généraux en corollaires.

3.4 Théorème *Pour tout $\psi \in \mathcal{SPEM}$ et pour toute* rs $\in$ LC-RVE*, si ψ possède une instance de* s-1-1 *calculable en temps linéaire, alors ψ possède une instance de* rs *calculable en temps linéaire.*

Preuve. Soit rs $= (\mathcal{R}, P) \in$ LC-RVE ; soient $\psi \in \mathcal{SPEM}$ et s une instance dans ψ de s-1-1 calculable en temps linéaire. Il est possible de construire directement une instance effective de rs dans ψ, cependant, pour alléger la présentation de la preuve, nous procédons par étapes. Nous commençons par construire une instance de s-2-1 dans ψ, puis une instance de 1-pkrt. Chacune de ces instances consiste en *un seul* appel à la fonction s, avec des arguments calculés à partir des arguments originaux en temps linéaire, et est donc calculable en temps linéaire. L'instance de 1-pkrt a une certaine injectivité que nous exploitons par la suite.

Comme ψ est effectif et maximal, il existe un ψ-programme a calculant $\lambda\langle\langle p, d_1, d_2\rangle, x\rangle.\psi_p(\langle d_1, d_2, x\rangle)$. Si on définit $t \stackrel{\mathrm{d}}{=} \lambda p, d_1, d_2.s(a, \langle p, d_1, d_2\rangle)$, on a que pour tout p, d_1 et d_2,

$$\begin{aligned}\psi(t(p, d_1, d_2)) = \psi(s(a, \langle p, d_1, d_2\rangle)) = \lambda x.\psi_a(\langle\langle p, d_1, d_2\rangle, x\rangle) = \\ = \lambda x.\psi_p(\langle d_1, d_2, x\rangle),\end{aligned}$$

ce qui revient à dire que t est une instance de $\mathsf{s\text{-}2\text{-}1}$ dans ψ. Comme s est calculable en temps linéaire, comme a est fixe et comme $\lambda p, d_1, d_2.\langle p, d_1, d_2\rangle$ est calculable en temps linéaire, il s'ensuit que t est aussi calculable en temps linéaire.

Comme $\psi \in \mathcal{SPEM}$ et $\mathsf{s\text{-}1\text{-}1} \in \mathsf{UTIL}(\psi)$, ψ est acceptable, et donc par le Théorème 2.69, $\mathsf{krt} \in \mathsf{UTIL}(\psi)$. Par krt dans ψ, il existe un ψ-programme e satisfaisant :

$$\psi_e = \lambda\langle p, d, x\rangle. \begin{cases} 1, & \text{si (i) } (\exists d' < d)[t(e,p,d') = t(e,p,d)], \\ 0, & \text{si (ii) } (\exists d' \mid d < d' \leq x)[t(e,p,d') = t(e,p,d)], \\ \psi_p(\langle t(e,p,d), d, x\rangle), & \text{autrement,} \end{cases} \tag{3.1}$$

où les conditions (i) et (ii) sont vérifiées dans cet ordre.

Le lecteur remarquera une similitude entre l'équation (3.1) et les constructions classiques d'une fonction de rembourrage dans un SP acceptable (voir par exemple [MY78]).

Notons d'abord que pour tout p, la fonction $\lambda d.t(e,p,d)$ est injective : en effet, supposons le contraire pour un certain p. Soit alors d le plus petit entier pour lequel il existe $d' > d$ tel que $t(e,p,d) = t(e,p,d')$. Par définition de $\mathsf{s\text{-}2\text{-}1}$, cela signifie que pour tout x, $\psi_e(\langle p, d, x\rangle) = \psi_e(\langle p, d', x\rangle)$. Or, par (3.1), $\psi_e(\langle p, d, d'\rangle) = 0 \neq 1 = \psi_e(\langle p, d', d'\rangle)$, une contradiction.

Ceci entraîne que pour tout p, d et x, ni l'une ni l'autre des conditions (i) et (ii) de (3.1) n'est satisfaite, et donc,

$$(\forall p, d, x)[\psi_e(\langle p, d, x\rangle) = \psi_p(\langle t(e,p,d), d, x\rangle)]. \tag{3.2}$$

Définissons maintenant $r \stackrel{\mathrm{d}}{=} \lambda p, d.t(e,p,d)$. De (3.2) et du fait que t est une instance de $\mathsf{s\text{-}2\text{-}1}$ dans ψ, on déduit que pour tout p et d,

$$\begin{aligned} \psi(r(p,d)) = \psi(t(e,p,d)) = \lambda x.\psi_e(\langle p,d,x\rangle) = \lambda x.\psi_p(\langle t(e,p,d), d, x\rangle) = \\ = \lambda x.\psi_p(\langle r(p,d), d, x\rangle), \end{aligned}$$

en d'autres mots, que r est une instance de $1\text{-}\mathsf{pkrt}$ dans ψ. De plus, comme pour tout p, la fonction $\lambda d.t(e,p,d)$ est injective, clairement pour tout p, $\lambda d.r(p,d)$ est injective. On note aussi que, t étant calculable en temps linéaire et e étant fixe, r est calculable en temps linéaire.

Soit m le nombre d'entrées de rs. Soient Θ un opérateur récursif déterminant rs, et Q un prédicat linéairement co-fini $(m+1)$-aire tel que pour toute

fonction m-aire f,

$$[f \text{ injective } \& (\forall p_1, \ldots, p_m) Q(p_1, \ldots, p_m, f(p_1, \ldots, p_m))] \Rightarrow P(f). \quad (3.3)$$

Soit k tel que pour tout $p_1, \ldots, p_m$, il existe au plus k valeurs de x pour lesquelles $\neg Q(p_1, \ldots, p_m, x)$. Comme ψ a puissance de calcul maximale et est exécutable, il existe un ψ-programme b correspondant à l'algorithme suivant :

Entrée : $\langle q, d, x \rangle$
Algorithme pour ψ_b **:**
1. Trouver $p_1, \ldots, p_m$ tels que $\langle p_1, \ldots, p_m \rangle = \lfloor d/(k+1) \rfloor$.
2. Par queue de colombe, calculer $\Theta(\psi, p_1, \ldots, p_m, q, x)(y+1)$ pour tout $y \in N$. Dès qu'un résultat est obtenu pour un certain $y_0 + 1$, retourner ce y_0.

□ **Algorithme**

Soit maintenant f la fonction calculée par l'algorithme suivant :

Entrée : $(p_1, \ldots, p_m)$
Algorithme pour f **:**
1. $d \leftarrow (k+1) \cdot \langle p_1, \ldots, p_m \rangle$.
2. $n \leftarrow (\mu z)[Q(p_1, \ldots, p_m, r(b, d+z))]$.
3. Retourner $r(b, d+n)$.

□ **Algorithme**

Comme $\lambda d.r(b,d)$ est injective, et comme pour tout $p_1, \ldots, p_m$ il existe au plus k valeurs de x pour lesquelles $\neg Q(p_1, \ldots, p_m, x)$, l'Étape 2 se termine toujours avec $n \leq k$. Aussi, r étant calculable en temps linéaire, Q étant décidable en temps linéaire, $\langle \cdot, \cdot \rangle$ étant calculable en temps linéaire et k étant fixe, on vérifie que chaque étape de l'algorithme prend un temps linéaire, et que donc, f est calculable en temps linéaire.

Encore parce que $\lambda d.r(b,d)$ est injective, et par l'algorithme pour f, on a que f est injective, et que pour tout $p_1, \ldots, p_m$, $Q(p_1, \ldots, p_m, f(p_1, \ldots, p_m))$. Donc, par (3.3), $P(f)$.

D'autre part, par l'algorithme pour f et par le fait que le n trouvé à l'Étape 2 ne dépasse jamais k, on a que pour tout $p_1, \ldots, p_m$, $f(p_1, \ldots, p_m)$ est de la forme $r(b, \langle p_1, \ldots, p_m \rangle \cdot (k+1) + n)$ pour un certain $n \leq k$. Par le fait que r est une instance de 1-pkrt dans ψ et par la sémantique du ψ-programme b, on déduit que pour chaque $p_1, \ldots, p_m$, le ψ-programme $f(p_1, \ldots, p_m)$ calcule la fonction partielle correspondant à l'algorithme suivant :

Entrée : x
Algorithme pour $\psi(f(p_1, \ldots, p_m))$ **:**
Par queue de colombe, calculer $\Theta(\psi, p_1, \ldots, p_m, f(p_1, \ldots, p_m), x)(y+1)$ pour tout $y \in N$. Dès qu'un résultat est obtenu pour un certain $y_0 + 1$, retourner ce y_0. □ **Algorithme**

Par la Proposition 2.17, nous concluons que f est une instance de rs dans ψ. □ **Théorème 3.4**

À notre connaissance, le Théorème 3.4 fournit les meilleures constructions d'instances de remb-inf et remb-inf-inj à partir d'une instance de s-1-1 jamais présentées dans la littérature. Il améliore en tout cas la construction de [Roy87] (le Lemme 1.4.3.14) qui ne garantit pour remb-inf qu'une instance élémentaire récursive à partir d'une instance de s-1-1 calculable en temps linéaire (Fait 3.3 (d)).

3.5 Corollaire *Il existe une opération ADHOC sur les classes de fonctions pour laquelle $ADHOC(\mathcal{C}) \subseteq \mathcal{C}$ dès que $\mathcal{C}$ est l'une de $\mathcal{L}intime$, $\mathcal{P}time$, $\mathcal{E}xptime$, $\mathcal{E}lm\mathcal{R}ec$ et $\mathcal{P}rim\mathcal{R}ec$, et telle que pour tout $\psi \in \mathcal{SPEM}$, pour toute* rs $\in$ LC-RVE*, et pour toute classe de fonctions $\mathcal{C}$, si ψ possède une instance de* s-1-1 *dans $\mathcal{C}$, alors ψ possède une instance de* rs *dans $ADHOC(\mathcal{C})$.*

Ce corollaire améliore le résultat de Royer pour les RS de VRAI-RIC dans les SP ayant une instance de s-1-1 dans $\mathcal{E}xptime$ (Fait 3.3 (c)). Il améliore aussi le résultat de Royer pour les RS de VRAI-RIC dans les SP ayant une instance de s-1-1 dans $\mathcal{P}time$ (Fait 3.3 (a)), puisqu'à partir d'une instance de s-1-1 calculable en temps $O(n^k)$, la construction de Royer donne en général des instances calculables en temps $O(n^{mk})$, où m est le nombre d'entrées de la RS, alors que la nôtre donne toujours des instances calculables en temps $O(n^k)$.

Royer a aussi étudié MR-RIC en relation avec mono-s-1-1 (Définition 3.2). Nous avons fait la preuve du Théorème 3.4 légèrement plus compliquée que nécessaire, de façon à en tirer certains corollaires qui améliorent des résultats de Royer sur ce sujet. Posons d'abord les définitions suivantes :

3.6 Définition (a) Soit $m > 0$. Un prédicat sur les fonctions m-aires P est dit *prédicat LCR* ssi il existe Q, un prédicat LC sur les fonctions

m-aires tel que pour toute fonction m-aire f,

$$[f \text{ strictement croissante en chaque argument } \& \; Q(f)] \Longrightarrow P(f).$$

(b) $\mathsf{LCR\text{-}RVE} \overset{\mathrm{d}}{=} \{(\mathcal{R}, P) \mid (\mathcal{R}, P) \in \mathsf{RVE} \text{ et } P \text{ est un prédicat LCR}\}$.

(c) $\mathsf{s\text{-}1\text{-}1\text{-}cr} \overset{\mathrm{d}}{=} (\mathcal{R}, P)$, où $\mathcal{R}$ est telle que $\mathsf{s\text{-}1\text{-}1} = (\mathcal{R}, \mathit{VRAI})$, et pour toute f 2-aire, $P(f) \Longleftrightarrow f$ est strictement croissante en chaque argument.

Informellement, s-1-1-cr a la même "sémantique" que s-1-1, mais exige de ses instances d'être strictement croissantes en chaque argument. C'est un simple exercice routinier de montrer que la classe des prédicats LCR contient strictement celle des prédicats MR. En particulier, LCR-RVE contient LC-RVE et toutes les RVE à prédicat MR (ce qui inclut MR-RIC).

3.7 Corollaire *Pour tout $\psi \in \mathcal{SPEM}$ et pour toute* rs $\in$ LCR-RVE*, si ψ possède une instance de* s-1-1-cr *calculable en temps linéaire, alors ψ possède une instance de* rs *calculable en temps linéaire.*

Preuve. Il suffit de remarquer que dans la preuve du théorème, si l'instance de s-1-1 est au départ strictement croissante en chaque argument (i.e., si elle est en fait une instance de s-1-1-cr), alors, l'instance de rs construite est elle aussi strictement croissante en chaque argument, en plus d'être injective et calculable en temps linéaire. Un facteur clé pour cet état de choses est que pour tout m, la fonction $\lambda p_1, \ldots, p_m . \langle p_1, \ldots, p_m \rangle$ est elle-même strictement croissante en chaque argument. □

Comme précédemment, on a aussi le corollaire :

3.8 Corollaire *Il existe une opération $ADHOC$ sur les classes de fonctions pour laquelle $ADHOC(\mathcal{C}) \subseteq \mathcal{C}$ dès que $\mathcal{C}$ est l'une de $\mathcal{L}intime$, $\mathcal{P}time$, $\mathcal{E}xptime$, $\mathcal{E}lm\mathcal{R}ec$ et $\mathcal{P}rim\mathcal{R}ec$, et telle que pour tout $\psi \in \mathcal{SPEM}$, pour toute* rs $\in$ LCR-RVE*, et pour toute classe de fonctions $\mathcal{C}$, si ψ a une instance de* s-1-1-cr *dans $\mathcal{C}$, alors il a une instance de* rs *dans $ADHOC(\mathcal{C})$.*

Ce corollaire améliore les Faits 3.3 (b) et (c) (partie "respectivement") de la même façon que le Corollaire 3.5 améliorait les Faits 3.3 (a) et (c). En plus,

cependant, la RS de départ utilisée par Royer (mono-s-1-1) est *a priori* plus "forte" que la nôtre (s-1-1-cr).

Que peut-on dire sur la complexité des instances des RS dans LCR-RVE dans un SP dont on connaît la complexité d'une instance de s-1-1 (et non s-1-1-cr) ? La construction suivante, qui corrige une construction de Royer [Roy87, Lemme 1.4.3.15] (résultat dont nous n'avons pas encore fait mention), nous donne un élément de réponse, peut-être un peu décevant.

3.9 Théorème *Pour tout $\psi \in \mathcal{SPEM}$, si ψ possède une instance de* s-1-1 *dans $\mathcal{P}time$, alors ψ possède une instance de* s-1-1-cr *dans $\mathcal{E}lm\mathcal{R}ec$.*

Preuve. Soit $\psi \in \mathcal{SPEM}$ et soit $s \in \mathcal{P}time$ une instance de s-1-1 pour ψ. Par le Corollaire 3.5, il existe $r \in \mathcal{P}time$ une instance de remb-inf pour ψ. Nous définissons une fonction t dont nous montrons qu'elle est une instance de s-1-1-cr pour ψ, puis nous argumentons sommairement qu'elle est calculable en espace borné par une fonction élémentaire récursive (t est en fait une instance de mono-s-1-1 pour ψ). Par [MY78, Théorème 1.10.6], le théorème sera établi. Voici la définition de t :

$$t \stackrel{\mathrm{d}}{=} \lambda p, x. \begin{cases} s(0,0) & \text{si } p = x = 0, \\ r(s(p,x), n) & \text{autrement, où } n = \\ & \quad (\mu m)[r(s(p,x), m) > t(q,y), \\ & \quad\quad \text{où } \langle q,y \rangle = \langle p,x \rangle - 1]. \end{cases}$$

On remarque pour tout p et x, $t(p,x)$ est de la forme $r(s(p,x), n)$ pour un certain n, et donc la sémantique de s-1-1 est respectée. D'autre part, pour tout q et y, on a clairement que $t(p,x) > t(q,y)$, où $\langle p,x \rangle = \langle q,y \rangle + 1$, ce qui entraîne que pour tout p, x, q et y, si $\langle p,x \rangle > \langle q,y \rangle$, alors $t(x,y) > t(q,y)$. Comme $\langle \cdot, \cdot \rangle$ est strictement croissante en chaque argument, on a que pour tout p et x, $t(p+1,x)$ et $t(p,x+1)$ sont toutes deux strictement supérieures à $t(p,x)$; t est donc bien une instance de s-1-1-cr pour ψ.

La définition de t suggère directement un algorithme pour la calculer. Nous argumentons que cet algorithme peut être implanté sur une machine de Turing fonctionnant en espace borné par une fonction élémentaire récursive.

Comme s et r sont dans $\mathcal{P}time$, il existe $\langle p_0, x_0 \rangle > 0$ et $e \geq 2$ tels que pour tout $\langle p,x \rangle \geq \langle p_0, x_0 \rangle$,

$$|s(p,x)| \leq |p,x|^e \ \& \ |r(p,x)| \leq |p,x|^e. \tag{3.4}$$

Notons également que, par un raisonnement immédiat, e peut être choisi de telle sorte que l'espace requis pour *calculer* les valeurs $s(p,x)$ et $r(p,x)$ à partir de p et x est lui aussi borné par $|p,x|^e$.

Comme pour tout p, $\lambda n.r(p,n)$ est injective, il s'ensuit que pour tout $\langle p,x\rangle \geq \langle p_0,x_0\rangle$, si on pose $\langle q,y\rangle = \langle p,x\rangle - 1$, alors *au moins une* des valeurs $r(s(p,x),n)$, $n \leq t(q,y)+1$, est supérieure à $t(q,y)$; autrement dit, $s(p,x)$ doit être rembourré avec une valeur d'*au plus* $t(q,y)+1$ pour devenir une valeur acceptable pour $t(p,x)$. Maintenant, $|t(q,y)+1| \leq |t(q,y)|+1$, et donc, par (3.4),

$$|t(p,x)| \leq (|p,x|^e + |t(q,y)| + 1)^e. \tag{3.5}$$

Notons que, comme on peut réutiliser l'espace pour les calculs successifs de $r(s(p,x),0)$, $r(s(p,x),1)$, ..., un espace d'au plus un multiple constant de $(|p,x|^e + |t(q,y)|+1)^e$ suffit pour *calculer* $t(p,x)$ à partir de $t(q,y)$, p et x.

Posons $L(\langle p_0,x_0\rangle) \stackrel{\mathrm{d}}{=} \max(2, |t(p_0,x_0)|)$ et récursivement, pour tout $\langle p,x\rangle > \langle p_0,x_0\rangle$, $L(\langle p,x\rangle) \stackrel{\mathrm{d}}{=} (|p,x|^e + L(\langle p,x\rangle - 1)+1)^e$. Par (3.5), il est clair que $|t(p,x)| \leq L(\langle p,x\rangle)$ dès que $\langle p,x\rangle \geq \langle p_0,x_0\rangle$ (on peut laisser L indéfinie sur les valeurs inférieures à $\langle p_0,x_0\rangle$).

Notons que, comme L est éventuellement non-décroissante, et comme on peut réutiliser l'espace pour les calculs successifs de $t(0,0)$, $t(0,1)$, ..., un espace d'au plus un multiple constant de $L(\langle p,x\rangle)$ suffit pour *calculer* la valeur $t(p,x)$ à partir de p et x dès que $\langle p,x\rangle \geq \langle p_0,x_0\rangle$. Autrement dit, L borne presque partout et à une constante près l'espace utilisé par notre algorithme.

Il est facile de montrer que L majore $|p,x|^e+1$ presque partout, i.e., qu'il existe un $\langle p_1,x_1\rangle \geq \langle p_0,x_0\rangle$ tel que pour tout $\langle p,x\rangle \geq \langle p_1,x_1\rangle$, $L(\langle p,x\rangle) \geq |p,x|^e+1$. Posons $LL(\langle p_1,x_1\rangle) \stackrel{\mathrm{d}}{=} L(\langle p_1,x_1\rangle)$ et récursivement, pour tout $\langle p,x\rangle > \langle p_1,x_1\rangle$, $LL(\langle p,x\rangle) \stackrel{\mathrm{d}}{=} (2 \cdot LL(\langle p,x\rangle - 1))^e$. En comparant les définitions de L et LL, on constate que, dès que $\langle p,x\rangle \geq \langle p_1,x_1\rangle$, $L(\langle p,x\rangle) \leq LL(\langle p,x\rangle)$.

Maintenant, si on pose $c \stackrel{\mathrm{d}}{=} L(\langle p_1,x_1\rangle)$, on tire facilement de la définition de LL que

$$LL(\langle p,x\rangle) \leq (2c)^{e^{\langle p,x\rangle+1}}$$

dès que $\langle p,x\rangle \geq \langle p_1,x_1\rangle$. Comme $\langle p,x\rangle \leq 2^{|\langle p,x\rangle|}$ et $|\langle p,x\rangle| \leq 2 \cdot |p,x|$, LL

est clairement élémentaire récursive en $|p, x|$, et donc, L l'est aussi. $\square$

L'énoncé du Lemme 1.4.3.15 de Royer promet une instance de mono-s-1-1 à partir d'une instance de s-1-1. La construction qui sert à établir ce lemme donne bien une instance de s-1-1-cr, mais pas nécessairement une de mono-s-1-1. Nous avons eu ici la situation contraire, où l'énoncé promet une instance de s-1-1-cr, mais où la construction fournit en fait une instance de mono-s-1-1. Nous avons utilisé cette construction non pas dans le but expresse d'obtenir une instance de mono-s-1-1, mais parce qu'elle s'analyse plus facilement que la construction de Royer.

En fait, l'énoncé du Lemme 1.4.3.15 de [Roy87] annonce que tout SP dans $\mathcal{SPEM}$ possédant une instance *élémentaire récursive* de s-1-1 posséderait une instance élémentaire récursive de s-1-1-cr (même de mono-s-1-1). Cependant, nous avons relevé une erreur dans l'analyse de la construction de Royer, et en réalité, sa construction ne permet pas d'affirmer beaucoup plus que notre Théorème 3.9, à première vue du moins. [1] Il ne nous surprendrait pas outre mesure que l'énoncé de [Roy87, Lemme 1.4.3.15] soit inexact, en ce sens qu'il existe un SP acceptable possédant une instance élémentaire récursive de s-1-1, mais aucune instance élémentaire récursive de s-1-1-cr. Nous n'avons cependant pas essayé de construire un tel SP.

Le Théorème 3.9 et le Corollaire 3.8 ont comme conséquence immédiate ceci :

3.10 Corollaire *Pour tout $\psi \in \mathcal{SPEM}$ et toute* rs $\in$ LCR-RVE*, si ψ possède une instance de* s-1-1 *dans $\mathcal{P}time$, alors il possède une instance de* rs *dans $\mathcal{E}lm\mathcal{R}ec$.*

3.2.2 Un résultat négatif

Par opposition aux résultats de bornes supérieures présentés ci-dessus, nous avons maintenant un résultat négatif pour *l'ensemble* de CC-RVE. D'abord, comme d'habitude, une définition.

3.11 Définition Une classe de fonctions partielles récursives $\mathcal{C}$ est dite *r.é.* ssi il existe une fonction récursive g telle que l'ensemble des fonctions par-

1. Nous avons rapporté cette erreur à Royer, qui l'a reconnue.

tielles récursives calculées par les machines de Turing $M_{g(i)}$, $i \in N$, coïncide avec $\mathcal{C}$. On dit alors que g *énumère* $\mathcal{C}$.

On note qu'une classe r.é. de fonctions partielles peut contenir des fonctions partielles d'arités différentes. Il est bien connu que chacune de $\mathcal{L}intime$, $\mathcal{P}time$, $\mathcal{E}xptime$, $\mathcal{E}lm\mathcal{R}ec$ et $\mathcal{P}rim\mathcal{R}ec$ sont r.é.

3.12 Théorème *Pour toute classe r.é. $\mathcal{C}$ de fonctions* récursives*, il existe une* $\mathsf{rs} \in \mathsf{CC\text{-}RVE}$ *telle qu'*aucun *SP ne possède une instance de* rs *dans $\mathcal{C}$. (En particulier, même les SP dans $\mathcal{SPEM}$ possédant une instance de* $\mathsf{s\text{-}1\text{-}1}$ *calculable en temps linéaire ne possèdent pas d'instance de* rs *dans $\mathcal{C}$.)*

Preuve. Nous montrons que pour toute classe r.é. $\mathcal{C}$ de fonctions récursives, le prédicat sur les fonctions unaires $P \stackrel{\mathrm{d}}{=} \lambda f.[f \notin \mathcal{C}]$ est un prédicat CC. Alors, $(\mathcal{R}, P)$, où $\mathcal{R}(\psi, p, q)$ est vrai pour tout ψ, p et q, est la RS désirée.

Pour établir que P est un prédicat CC, il suffit, par la Définition 2.66, d'exhiber un prédicat sur les séquences cumulativement co-fini Q tel que pour toute fonction unaire f,

$$[(\forall x) Q(f(0) : f(1) : \cdots : f(x))] \Longrightarrow P(f)$$

(on utilise l'identité comme fonction de pairage unaire). On construit un tel Q par diagonalisation.

Soit g une fonction récursive qui énumère $\mathcal{C}$. Définissons Q par l'algorithme suivant :

> **Entrée :** σ (une séquence d'entiers)
> **Algorithme pour Q :**
> 1. Si $\ell(\sigma) = 0$, retourner vrai.
> 2. $i \leftarrow \ell(\sigma) - 1$.
> 3. Si $M_{g(i)}$ n'est pas unaire, retourner vrai.
> 4. Si $\sigma_{\ell(\sigma)} \neq M_{g(i)}(i)$, retourner vrai ; autrement, retourner faux.
>
> □ **Algorithme**

On vérifie aisément que Q est cumulativement co-fini. On vérifie également que pour toute f unaire et tout x, $Q(f(0) : f(1) : \cdots : f(x)) \Rightarrow [M_{g(x)}$

n'est pas unaire ou $f(x) \neq M_{g(x)}(x)] \Rightarrow [f$ diffère de la fonction calculée par $M_{g(x)}]$. Si $Q(f(0) : f(1) : \cdots : f(x))$ est vrai pour tout x, c'est donc que f n'est aucune des fonctions calculées par les machines de Turing $M_{g(x)},\ x \in N$; autrement dit, $f \notin \mathcal{C}$ et donc, $P(f)$. □

3.2.3 Remarques

Nous notons que la fonction de pairage utilisée dans la définition de s-1-1 n'a aucune incidence sur les résultats de cette section. Nous avons utilisé une fonction de pairage calculable en temps linéaire, mais n'importe quelle autre fonction de pairage ferait l'affaire, *à la condition* d'utiliser une fonction de pairage calculable en temps linéaire (même si elle s'avérait différente de celle dans la définition de s-1-1) dans la définition de f dans la preuve du Théorème 3.4.

Le contraste entre les Théorèmes 3.4 et 3.12 est frappant et suggère que les RS qui ne sont *pas* garanties efficacement par s-1-1 sont dans cette situation parce que leur contrainte textuelle est “trop grande”. Dans cette optique, la classe LC-RVE se présente comme une candidate au moins raisonnable de l'ensemble des RS dont on peut s'attendre qu'elles soient récursivement satisfaisables *efficacement* dans n'importe quel “bon” SP acceptable, puisque leur contrainte textuelle n'impose aucune complexité d'implantation indue. En fait, on pourrait informellement *définir* un “bon” SP acceptable comme en étant un où toutes les RS de LC-RVE sont récursivement satisfaisables efficacement. Le Théorème 3.4 nous dit alors que tout SP ayant une instance “facile” de s-1-1 est “bon”.

La complexité des instances des RS de LC-RVE peut nous aider à “jauger” la complexité de la “tâche de programmer” dans un SP. Si une RS dans LC-RVE n'est que difficilement implantable dans un SP, la faute en est au SP, puisque cette RS *est* implantable efficacement dans certains SP. Par contre, si une RS qui n'est *pas* dans LC-RVE n'est que difficilement implantable dans un SP, on peut soupçonner que la difficulté d'implantation est intrinsèque, et non pas une particularité du SP.

3.3 L'interrelation comp $\models_m$ s-1-1

Dans cette section, nous considérons l'aspect complexité de l'interrelation comp $\models_m$ s-1-1. Certains des résultats de cette section ont paru dans [Mar89].

Précédemment, la meilleure (en fait, la seule) borne supérieure sur la complexité d'une instance de s-1-1 dans un SP possédant une instance dans $\mathcal{L}$*intime* de comp était donnée par une construction de Machtey et Young [MY78, Théorème 3.1.2], et était double-exponentielle. À deux reprises dans la littérature, l'intuition a été exprimée que cette borne supérieure ne pouvait pas être sensiblement améliorée [MWY78, Roy87]. Nous donnons une borne supérieure polynomiale et la montrons optimale. Nous déduisons différents corollaires, dont celui qu'il existe un SP *acceptable* possédant un instance de comp calculable en temps polynomial mais aucune instance de s-1-1 calculable en temps sous-exponentiel. Il ne semble donc pas que comp ait le même caractère fondamental que s-1-1 en ce qui a trait à la puissance expressive, "qualifiée" par la complexité d'instances des RS garanties.

Le premier résultat principal de cette section est notre Théorème 3.13. Nous y montrons que tout SP maximal possédant une instance de comp calculable en temps linéaire possède une instance de s-1-1 calculable en temps polynomial, avec le degré du polynôme essentiellement donné par le logarithme en base 2 de la constante linéaire de l'instance de comp.

La construction de Machtey et Young d'une instance effective de s-1-1 à partir d'une instance effective de comp est esquissée en parallèle avec la preuve de notre Théorème 3.13. Une analyse de cette construction révèle que si l'instance de départ de comp est calculable en temps linéaire, alors, l'instance construite de s-1-1 est calculable en temps double-exponentiel ; par contre, un simple argument montre que l'instance construite *requiert* parfois ce temps de calcul presque partout (cette borne inférieure s'applique seulement à l'instance construite, et non pas à n'importe quelle instance de s-1-1). La construction est donc inefficace. (Il faut noter ici que Machtey et Young ne se préoccupaient pas de questions de complexité, simplement de calculabilité.)

Comme nous avons dit, à deux reprises dans la littérature, l'intuition a été exprimée que cette construction ne pouvait pas être améliorée sensiblement. Dans les deux cas, cette intuition était basée sur celle qu'il existait un SP acceptable dans lequel *toute* instance de s-1-1 était beaucoup plus complexe

qu'une instance de comp (mettons, la "plus facile"). Dans [MWY78] (p. 53), Machtey, Winklmann et Young écrivent qu'il ne serait pas contre-intuitif si la tâche de calculer n'importe quelle instance de s-1-1 s'avérait être (dans un certain SP) sensiblement plus complexe que celle de calculer *une* instance de comp, au point de rendre la construction de [MY78] quasi-optimale. Dans [Roy87] (pp. 39 et 152), Royer est plus spécifique et suggère qu'il est possible de construire un SP acceptable avec une instance de comp calculable en temps linéaire, mais aucune instance de s-1-1 calculable en temps sous-exponentiel.

Cette intuition est ici réfutée par notre Théorème 3.13. Cependant, notre autre résultat principal de cette section, le Théorème 3.18 confirme qu'il y a bien une différence entre les puissances *qualifiées* de s-1-1 et comp, même si, intuitivement, cette différence est moindre que prévue par Machtey, Winklmann et Young et Royer. Ce théorème énonce essentiellement que la borne supérieure donnée par notre Théorème 3.13 est optimale pour une grande famille de SP *acceptables*. En corollaire (Corollaire 3.23), nous exhibons un SP acceptable possédant une instance de comp calculable en temps polynomial, mais aucune instance de s-1-1 calculable en temps sous-exponentiel. Sous l'hypothèse que les calculs pratiquement "faisables" sont ceux qui peuvent être réalisés en temps polynomial, et sous l'hypothèse que la "tâche de programmer" d'un SP est représentée par ses instances des RS dans LC-RVE, la tâche de programmer dans un tel SP n'est pas pratiquement faisable (puisque s-1-1 $\in$ LC-RVE), bien qu'il ait une instance pratiquement calculable de comp. Par contraste, le Corollaire 3.5 nous dit que dès qu'un SP acceptable a une instance pratiquement calculable de s-1-1, la tâche d'y programmer est pratiquement faisable.

3.3.1 La borne supérieure

3.13 Théorème *Supposons que $\psi \in \mathcal{SPM}$ possède une instance de* comp *calculable partout en temps $qn + k$ pour un certain $q \geq 1$ et un certain k. Alors, si $q > 1$, ψ possède une instance de* s-1-1 *calculable en temps $O(n^{1+\lg q})$; si $q = 1$, ψ possède une instance de* s-1-1 *calculable en temps $O(n \log n)$.*

Preuve. Nous ne présentons que le cas $q > 1$; le cas $q = 1$ se démontre de façon analogue.

Soit $\psi \in \mathcal{SPM}$ et c une instance de comp dans ψ qui est calculable partout en temps $qn + k$ pour un certain $q > 1$ et un certain k. Il sera pratique de supposer, sans perte de généralité, que $k \geq 1$. Nous décrivons un algorithme qui, utilisant c, calcule une instance de s-1-1 dans ψ, et s'exécute en temps $O(n^{1+\lg q})$.

À plusieurs égards, notre algorithme est similaire à celui de [MY78]. Sur entrée (p, x), nous construisons d'abord un "programme insérant x", i.e., un programme pour la fonction $\lambda z.\langle x, z\rangle$, puis nous composons ce programme à droite de p. Comme dans [MY78], nous construisons notre programme insérant x en composant ensemble des copies d'un nombre fini de "programmes de base". Notre construction diffère cependant quant au choix des programmes de base, et quant à la façon dont nous composons ensemble les copies de ces programmes. Informellement, chacune de ces différences nous débarrasse d'un niveau d'exponentielle.

Machtey et Young utilisent deux programmes de base, appelons-les p_0 et p_1, dont le premier insère un 0 à la gauche de son argument (i.e., calcule $\lambda z.\langle 0, z\rangle$) et dont le second augmente de 1 son premier argument sans toucher à l'autre (i.e., calcule $\lambda\langle n, z\rangle.\langle n + 1, z\rangle$). Un x étant donné, ils obtiennent leur programme insérant x en composant x copies de p_1 à gauche de p_0. Intuitivement, ce programme insérant x "construit" x en comptant de 0 à x. Par contraste, notre programme insérant x construit x bit à bit. Pour réaliser cela, nous utilisons quatre programmes de base, appelons-les p_0 à p_3, satisfaisant

$$\begin{aligned} \psi_{p_0} &= \lambda z.\langle 1, 0, z\rangle, \\ \psi_{p_1} &= \lambda\langle x, y, z\rangle.\langle 2x, y, z\rangle, \\ \psi_{p_2} &= \lambda\langle x, y, z\rangle.\langle 2x, y + x, z\rangle \text{ et} \\ \psi_{p_3} &= \lambda\langle x, y, z\rangle.\langle y, z\rangle. \end{aligned} \tag{3.6}$$

Nous obtenons alors notre programme insérant x en juxtaposant, de gauche à droite, d'abord le programme p_3, puis un de p_1 et p_2 pour chaque bit de x en commençant par le bit le plus significatif (p_1 si le bit est 0, et p_2 autrement), puis finalement p_0, et en composant tous ces programmes ensemble. Dans cette dernière étape, notre deuxième truc entre en jeu.

Machtey et Young composaient leurs programmes de base "séquentiellement", en ce sens qu'ils n'avaient en tout temps qu'un seul "programme insérant x en herbe" avec lequel était composée immédiatement toute nouvelle copie de programme de base devant être "juxtaposée" à celles déjà en place (dans les termes du paragraphe précédent). Au lieu de cela, nous tirons avantage

de l'associativité de la composition de fonctions, et composons nos copies de programmes de base suivant un patron d'arbre binaire (stratégie diviser-pour-régner). Nous "juxtaposons" d'abord littéralement toutes nos copies de programmes de base sur un ruban. Puis, nous les composons deux-à-deux, écrivant les programmes résultants consécutivement sur un autre ruban. Nous répétons cette opération jusqu'à ce qu'il ne nous reste qu'un seul programme. Ce programme est notre programme insérant x.

Il peut sembler curieux qu'une stratégie diviser-pour-régner s'avère profitable pour effectuer une série d'opérations associatives. En effet, peu importe la façon dont on s'y prenne, traiter k opérandes demande toujours $k-1$ opérations. Il faut se rappeler, cependant, que l'instance de comp c effectue une opération qui est *sémantiquement* associative, mais pas nécessairement *textuellement* associative. En effet, si i, j et k sont des ψ-programmes, alors les ψ-programmes $c(i, c(j,k))$ et $c(c(i,j),k)$, bien qu'équivalents, sont en général distincts. Ainsi donc, le "patron d'application" de c à une séquence de programmes, bien que n'ayant pas d'influence sur la *sémantique* du programme résultant, peut avoir une influence sur le *texte* de celui-ci, et en particulier, sur sa *longueur*. Nous exploitons ce phénomène ici. En composant nos copies de programmes de base suivant un patron d'arbre binaire, chaque copie est "passée dans c" le moins souvent possible, et la longueur des programmes intermédiaires ne croît pas trop vite.

Voici maintenant un peu plus précisément notre algorithme. Nous le décrivons en des termes proches du langage des machines de Turing, de façon à faciliter l'analyse subséquente de son temps d'exécution sur une machine de Turing.

> **Entrée :** (p, x), où p est un ψ-programme et d, un entier.
> **Algorithme :**
>
> Nous utilisons quatre ψ-programmes de base, appelés p_0 à p_3, qui satisfont (3.6) et une sous-routine pour la fonction c, que nous supposons s'exécuter en temps $qn+k$.
>
> Dans une phase d'initialisation, nous écrivons successivement sur un ruban de travail p_3, puis p_{b+1} pour chaque bit b de x (en commençant par le bit le plus significatif), puis p_0. (Deux programmes adjacents sont séparés par un symbole spécial.) Dénotons par n_0 le nombre de programmes écrits sur le ruban de travail pendant la phase d'initialisation (clairement, $n_0 = |x|+2$), et par $\pi_{0,j}$ le j-ième de ces programmes ($1 \leq j \leq n_0$).

Vient alors une phase itérative pendant laquelle les programmes sur le ruban de travail sont composés ensemble (à l'aide de notre sous-routine pour c) suivant un patron d'arbre binaire. À chaque itération de cette phase, le ruban de travail résultant de l'itération précédente (ou de la phase d'initialisation, s'il s'agit de la première itération) est utilisé comme ruban d'entrée de l'itération, et un nouveau ruban de sortie est produit. (Clairement, deux rubans de travail, servant alternativement de ruban d'entrée puis de ruban de sortie, suffisent pour implanter les rubans d'entrée et de sortie de toutes les itérations.)

Supposons qu'au moins b itérations aient lieu. Pour tout i satisfaisant $1 \leq i \leq b$, dénotons par n_i le nombre de programmes écrits sur le ruban de sortie pendant la i-ième itération, et par $\pi_{i,j}$ le j-ième de ces programmes ($1 \leq j \leq n_i$). Le traitement effectué pendant la i-ième itération ($1 \leq i \leq b$) est le suivant :

Successivement, pour chaque j satisfaisant $1 \leq j \leq \lfloor n_{i-1}/2 \rfloor$, nous calculons $c(\pi_{i-1,2j-1}, \pi_{i-1,2j})$, que nous écrivons sur le ruban de sortie, et qui constitue ainsi le programme $\pi_{i,j}$. Si n_{i-1} est pair, l'itération est terminée, autrement, nous copions le programme $\pi_{i-1,n_{i-1}}$ à la fin du ruban de sortie, où il devient connu sous le nom de $\pi_{i,\lceil n_{i-1}/2 \rceil}$.

La phase itérative se termine lorsqu'un ruban de sortie est produit qui ne contient qu'un seul programme. Supposons que la phase itérative se termine, et qu'elle se termine après b étapes. Nous pouvons alors désigner l'unique programme écrit pendant la b-ième itération par $\pi_{b,1}$.

Dans une phase finale, le ψ-programme $c(p, \pi_{b,1})$ est calculé, et produit comme sortie de l'algorithme.

□ **Algorithme**

Nous n'avons pas mentionné explicitement, mais devrons considérer dans l'estimation du temps d'exécution de l'algorithme, le fait que la tête du ruban produit au cours de la phase d'initialisation et de chaque itération doit être repositionnée au début du ruban afin que celui-ci puisse servir de ruban d'entrée pour l'itération suivante.

Il est clair qu'au moins une itération a lieu dans la phase itérative. Soient p et x fixés, et soit $i > 0$ tel qu'au moins i itérations ont lieu dans la phase itérative. Nous donnons à n_k et $\pi_{k,j}$ ($k \leq i$ et $1 \leq j \leq n_k$) la même signification que dans la description de l'algorithme.

Il est immédiat, d'après la description de l'algorithme, que $n_i = \lceil n_{i-1}/2 \rceil$ et que donc, la phase itérative se termine après exactement $b \overset{\mathrm{d}}{=} \lceil \lg n_0 \rceil$ itérations. Notre algorithme calcule donc une fonction totale.

Il est facile de montrer, par induction sur $|x|$, que

$$\psi(\pi_{0,1}) \circ \psi(\pi_{0,2}) \circ \cdots \circ \psi(\pi_{0,n_0}) = \lambda z.\langle x, z \rangle.$$

Observons aussi que, par associativité de la composition de fonctions, et par le fait que c est une instance de comp dans ψ,

$$\psi(\pi_{i,1}) \circ \psi(\pi_{i,2}) \circ \cdots \circ \psi(\pi_{i,n_i}) = \psi(\pi_{i-1,1}) \circ \psi(\pi_{i-1,2}) \circ \cdots \circ \psi(\pi_{i-1,n_{i-1}}).$$

On a donc que

$$\psi(\pi_{i,1}) \circ \psi(\pi_{i,2}) \circ \cdots \circ \psi(\pi_{i,n_i}) = \lambda z.\langle x, z \rangle.$$

En particulier, $\pi_{b,1}$ (l'unique programme produit à la dernière iteration), est un ψ-programme pour $\lambda z.\langle x, z \rangle$. Il est clair, alors, que notre algorithme calcule une instance de s-1-1 dans ψ.

Nous analysons maintenant le temps d'exécution de l'algorithme sur une machine de Turing. Dénotons respectivement par T_{init}, T_{loop}, T_{term}, $T_{loop,i}$ et T les temps d'exécution de la phase d'initialisation, de la phase itérative, de la phase finale, de la i-ième itération et de l'ensemble de l'algorithme. Aussi, pour i satisfaisant $0 \leq i \leq b$, posons

$$l_i \overset{\mathrm{d}}{=} \sum_{j=1}^{n_i} |\pi_{i,j}|.$$

Clairement, $l_0 \leq T_{init} \leq c_1 n_0$ pour une certaine constante c_1. L'observation suivante découle directement de nos hypothèses sur c et sur la sous-routine que nous utilisons pour la calculer.

3.14 Observation Pour tous a et b, $|c(a,b)| \leq q(|a| + |b|) + k$, et le calcul de $c(a,b)$ dans le cours de l'algorithme peut s'effectuer en temps $c_2(q(|a| + |b|) + k)$ pour une certaine constante c_2.

Nous affirmons d'abord que $l_i \leq q l_{i-1} + n_i k$ et que $T_{loop,i} \leq c_3(q l_{i-1} + n_i k)$ pour une certaine constante c_3 : en effet, ces affirmations découlent de

l'Observation 3.14 et du fait que la i-ième itération consiste à "passer dans c" (au plus) les programmes $\pi_{i-1,j}$, où j satisfait $1 \leq j \leq n_{i-1}$, cette opération étant réalisable par (au plus) n_i applications de c (le dernier programme est simplement copié si n_{i-1} est impair). (Il faut ici faire attention aux faits que la tête du ruban de sortie d'une itération doit être repositionnée à la fin de l'itération, et que deux programmes adjacents sur un ruban sont séparés par un symbole spécial. Cependant, nos affirmations sont quand même vérifiées, en partie parce que nous avons supposé $k \geq 1$.)

Ensuite, nous affirmons que $n_i \leq 2^{1-i} n_0$. En effet, rappelons-nous que $n_i = \lceil n_{i-1}/2 \rceil$. Si n_0 est une puissance 2, alors clairement $n_i = 2^{-i} n_0$. Autrement, observons que, comme $\lambda z. \lceil z/2 \rceil$ est non-décroissante, n_i n'excède sûrement pas la valeur qu'il aurait si l'on remplaçait n_0 par la plus petite puissance de 2 qui lui est supérieure. Formellement, cet argument se traduit par $n_i \leq 2^{-i} 2^{\lceil \lg n_0 \rceil}$. Comme le dernier facteur est plus petit que $2n_0$, notre affirmation est vérifiée.

Des première et troisième affirmations ci-dessus, on montre aisément (par induction sur i) que $l_i \leq q^i l_0 + 2kn_0 \cdot \sum_{j=1}^{i} 2^{-j} q^{i-j}$. Comme la sommation est moindre que q^i, et puisque $l_0 \leq c_1 n_0$, on obtient

$$l_i \leq c_4 n_0 q^i, \tag{3.7}$$

où $c_4 = c_1 + 2k$. Similairement, on déduit des deuxième et troisième affirmations que $T_{loop,i} \leq c_3 c_4 n_0 q^i$.

Nous sommes donc en mesure de poser

$$T_{loop} = \sum_{i=1}^{b} T_{loop,i} \leq c_3 c_4 n_0 \cdot \sum_{i=1}^{b} q^i.$$

Grâce à la familière identité géométrique, la dernière sommation devient

$$\frac{q^{b+1} - q}{q - 1} < \frac{q^2}{q-1} q^{\lg n_0} = \frac{q^2}{q-1} n_0^{\lg q}.$$

On obtient enfin $T_{loop} \leq c_5 n_0^{1+\lg q}$, où $c_5 = c_3 c_4 q^2/(q-1)$.

Pour estimer T_{term}, nous utilisons d'abord l'Observation 3.14 pour le borner supérieurement par $c_2(q(|p| + |\pi_{b,1}|) + k)$. Ensuite, utilisant le fait que $|\pi_{b,1}|$ est l_b et (3.7), on obtient

$$T_{term} \leq c_2 q \cdot |p| + c_2 c_4 q^2 n_0^{1+\lg q} + c_2 k.$$

Gardant à l'esprit que $n_0 \geq 2$ et $\lg q > 0$ en sommant T_{init}, T_{loop} et T_{term}, on obtient

$$T \leq c_6 \cdot |p| + c_6 n_0^{1+\lg q},$$

où $c_6 = c_1 + c_5 + c_2 q + c_2 c_4 q^2 + c_2 k$.

Or, $|x| \leq |p, x|$ and $|p| \leq |p, x|$. Donc, excluant les 2 cas où $|p, x| < 2$, nous avons

$$T \leq c_6(2q + 1) \cdot |p, x|^{1+\lg q}.$$

□ **Théorème 3.14**

Par les Corollaires 3.5 et 3.10, nous avons immédiatement :

3.15 Corollaire *Tout SP dans $\mathcal{SPEM}$ possédant une instance de* comp *dans $\mathcal{L}intime$ possède une instance dans $\mathcal{P}time$ de toute RS dans* LC-RVE, *et une instance dans $\mathcal{E}lm\mathcal{R}ec$ de toute RS dans* LCR-RVE.

La construction dans notre preuve du Théorème 3.13 est applicable à des instances de comp aussi complexes que l'on veut, mais un résultat vraiment général semble difficile à énoncer sans l'introduction d'une opération *ad hoc* sur les classes de fonctions. Dans le cas d'une instance de comp dans $\mathcal{P}time$, cependant, nous avons le corollaire suivant. (Rappelons que $\mathcal{E}xptime$ est la classe de fonctions calculables en temps $O(2^{p(n)})$ pour un certain polynôme p.)

3.16 Corollaire *Tout SP dans $\mathcal{SPM}$ possédant une instance de* comp *dans $\mathcal{P}time$ possède une instance de* s-1-1 *dans $\mathcal{E}xptime$.*

Preuve. Similaire à la preuve du théorème. Une analyse beaucoup moins fine suffit. □

Par le Corollaire 3.5, nous avons également :

3.17 Corollaire *Tout SP dans $\mathcal{SPEM}$ possédant une instance de* comp *dans $\mathcal{P}time$ possède une instance dans $\mathcal{E}xptime$ de toute RS dans* LC-RVE.

3.3.2 La borne inférieure

Nous montrons maintenant que notre construction d'une instance de s-1-1 dans la preuve du Théorème 3.13 est optimale pour un grand éventail de SP *acceptables* possédant une instance de comp dans $\mathcal{L}intime$. Plus précisément, pour tout rationnel $q \geq 1$,[2] nous exhibons un SP acceptable possédant une instance de comp calculable en temps $qn + k$, pour un certain k petit (qui ne dépend pas de q), mais dont *toutes* les instances de s-1-1 requièrent un temps de calcul dans $\Omega(n^{1+\lg q})$ ($\Omega(n \log n)$ si $q = 1$).

3.18 Théorème *Il existe un SP acceptable possédant une instance de* comp *calculable en temps $n + k$ pour un certain k petit, mais dont toutes les instances effectives de* s-1-1 *requièrent un temps de calcul dans $\Omega(n \log n)$ infiniment souvent. Pour tout rationnel $q > 1$, il existe un SP acceptable possédant une instance de* comp *calculable en temps $qn + k$ pour un certain k petit (qui ne dépend pas de q), mais dont toutes les instances effectives de* s-1-1 *requièrent un temps de calcul dans $\Omega(n^{1+\lg q})$ infiniment souvent.*

Preuve. Nous ne présentons que le cas $q > 1$; le cas $q = 1$ se démontre de façon analogue.

Soit $q > 1$ un rationnel. Nous construisons un SP ψ qui satisfait les conditions du théorème. L'exposé de la preuve est facilité si nous considérons les ψ-programmes comme étant des chaînes de symboles plutôt que des entiers. Le domaine de ψ sera donc toutes les chaînes finies sur l'alphabet $\{$[,], $*$, a, b, c$\}$. Avec seulement de très mineures additions (concernant la façon d'encoder les ψ-programmes en entiers), notre preuve pourrait fournir un SP de domaine N. Le k dans l'énoncé du théorème n'a pas à excéder 4 fois le nombre maximum de bits utilisés pour encoder les symboles de l'alphabet des ψ-programmes.

Dans notre discussion, toute variable utilisée pour dénoter un ψ-programme est implicitement de "type" chaîne. La longueur d'une chaîne a, dénotée $|a|$ est le nombre d'occurrences de symboles dans a. Si a et b sont des chaînes, $|a, b|$ dénote $|a| + |b|$.

---

2. Tous les résultats de cette section qui font mention d'un rationnel $q \geq 1$ sont en fait valides pour tout *réel* $q \geq 1$ tel que $\lambda n.\lfloor qn \rfloor + 1$ est pleinement constructible en temps [HU79]. Ceci inclut non seulement tout rationnel supérieur ou égal à 1, mais aussi certains irrationnels (voir [Marc]).

Comme les ψ-programmes sont des chaînes de symboles, une instance de comp pour ψ prend comme arguments deux chaînes de symboles, et une instance de s-1-1 prend comme arguments une chaîne de symboles et un entier. Pour parler de la complexité de ces fonctions, nous sous-entendrons une extension naturelle et immédiate de notre modèle de calcul à un alphabet d'entrée autre que $\{0, 1\}$. La *longueur de l'entrée* sera alors naturellement la somme des longueurs des deux arguments, où la longueur d'une chaîne est définie comme nous venons de dire.

Nous définissons maintenant ψ. La fonction g sera une instance "intrinsèque" de comp pour ψ, et nous en dirons plus long sur elle dans quelques instants.

3.19 Définition

$$\begin{array}{lcll}
\psi(\mathsf{a}) & \stackrel{\mathrm{d}}{=} & \lambda z.\langle 0, z\rangle, & \\
\psi(\mathsf{b}) & \stackrel{\mathrm{d}}{=} & \lambda\langle y, z\rangle.\langle y+1, z\rangle, & \\
\psi(\mathsf{c}) & \stackrel{\mathrm{d}}{=} & \lambda\langle p, x\rangle.\phi_p(x), & \\
\psi(g(a,b)) & \stackrel{\mathrm{d}}{=} & \psi_a \circ \psi_b, & \text{pour tout } a,\ b, \\
\psi(p) & \stackrel{\mathrm{d}}{=} & \lambda z.\uparrow, & \text{pour tout } p \notin \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \cup \mathbf{image}(g).
\end{array}$$

Supposons pour le moment que g est récursive et ne cause ni conflit ni circularité dans la Définition 3.19.

Clairement, par le fait que ϕ est exécutable et g récursive, ψ est un SP exécutable. Remarquons aussi que g est, par construction, une instance de comp pour ψ.

Observons que les programmes a et b et la fonction g nous permettent de réaliser la construction originelle de Machtey et Young d'une instance de s-1-1, et qu'il existe donc une fonction récursive s telle que pour tout p et x, $\psi_{s(p,x)} = \lambda z.\psi_p(\langle x, z\rangle)$. Alors, en particulier,

$$\psi_{s(\mathsf{c},x)} = \lambda z.\psi_{\mathsf{c}}(\langle x, z\rangle) = \lambda z.\phi_x(z) = \phi_x.$$

Donc, ψ a au moins un programme pour chaque fonction partielle récursive, et a donc puissance de calcul maximale. Comme il est exécutable, a puissance de calcul maximale et possède une instance effective de comp, il est acceptable. [3]

3. Une interprétation du fait que ψ a puissance de calcul maximale est que l'ensemble

Penchons-nous maintenant sur g. Tout d'abord, nous voulons que g soit calculable en temps $qn + k$ pour un certain k petit, car nous voulons que ψ ait une instance de comp calculable en cette limite de temps. Cependant, nous voulons aussi que g ait une sortie (i.e., une valeur) de longueur au moins qn. Appelons cette exigence "l'exigence d'une sortie de longueur qn", pour référence future. L'idée sous-jacente à cette exigence est de forcer les ψ-programmes qui sont le résultat de plusieurs applications de g à être longs. Nous verrons plus tard que pour toute instance s de s-1-1 en ψ, certains programmes dans **image**(s) *doivent* être le résultat de plusieurs applications de g. Si tous ces programmes sont longs, nous pourrons borner inférieurement le temps de calcul de s.

Bien sûr, nous voulons aussi que g soit récursive et n'introduise ni conflit ni circularité dans la Définition 3.19. Un premier essai de définition pourrait se

---

$\{\psi(\mathsf{a}), \psi(\mathsf{b}), \psi(\mathsf{c})\}$ constitue une "base" génératrice de toutes les fonctions partielles récursives unaires, en ce sens que sa fermeture sous la composition de fonctions coïncide avec $\mathcal{P}art\mathcal{R}ec\mathcal{U}n$. Lorsque mis au défi par l'auteur de démontrer qu'aucune base de moins de trois éléments ne peut engendrer $\mathcal{P}art\mathcal{R}ec\mathcal{U}n$, Stuart Kurtz [Kur89] trouva rapidement le contrexemple suivant (en essence). Dénotons par $[x]$ la chaîne (sur $\{0,1\}$) donnant la représentation binaire minimale de l'entier x, et par $v(\sigma)$, la valeur de la chaîne σ interprétée comme un entier binaire. Notons que $\lambda x, i.v([2x]10^{i+1}11)$ est injective et strictement croissante en chaque argument. Définissons maintenant α et β comme suit :

$$
\begin{array}{lcll}
\alpha(n) & \stackrel{\mathrm{d}}{=} & 2n & \text{pour tout } n, \\
\beta(v([2x]10^{i+1}11)) & \stackrel{\mathrm{d}}{=} & \phi_i(x) & \text{pour tout } (x,i), \\
\beta(n) & \stackrel{\mathrm{d}}{=} & 2n+1 & \text{si } n \text{ n'égale } v([2x]10^{i+1}11) \text{ pour aucun } (x,i).
\end{array}
$$

Clairement, α et β sont partielles récursives. On vérifie facilement que pour tout i,

$$\phi_i = \beta^3 \circ \alpha^{i+1} \circ \beta \circ \alpha. \tag{3.8}$$

Ainsi, $\{\alpha, \beta\}$ engendre $\mathcal{P}art\mathcal{R}ec\mathcal{U}n$. De plus, (3.8) suggère une procédure uniforme de traduction de ϕ-programmes en (descriptions de) séquences de α et β équivalentes. Notons qu'avec une représentation naturelle des séquences de α et β (par exemple, des chaînes de a et de b), cette procédure prend un temps exponentiel. Cependant, comme corollaire à notre Théorème 3.13, il existe une procédure de traduction prenant temps $O(n \log n)$ et, en fait, il est facile de voir qu'il en existe une fonctionnant en temps linéaire. Cette situation est très réminiscente du fait, souligné plus bas, que l'algorithme le plus efficace pour une instance de s-1-1 pour ψ n'est *pas* celui qui est naturellement suggéré par la définition de ψ.

On montre par un raisonnement immédiat qu'aucune base d'un élément n'engendre $\mathcal{P}art\mathcal{R}ec\mathcal{U}n$. (Une fonction totale ne peut engendrer que des fonctions totales et une fonction non-totale ne peut engendrer que des fonctions non-totales.)

lire comme suit :

$$g(a,b) \stackrel{\mathrm{d}}{=} [a][b]\underbrace{*\cdots\cdots\cdots\cdots\cdots*}_{\lfloor (q-1)|a,b| \rfloor \text{ fois}}.$$

Cette fonction satisfait "l'exigence d'une sortie de longueur qn" en utilisant le symbole "$*$" comme rembourrage. Nous verrons dans un instant qu'elle est calculable en temps $\lfloor qn \rfloor + k$ pour un certain k petit ; de plus, elle ne cause aucune circularité dans la Définition 3.19, puisque $|g(a,b)| > \max(|a|,|b|)$ pour tout a et b.

Cependant, elle n'est pas injective, et peut causer des conflits dans la définition de ψ. En effet, avec $a =$ "[a][a]", $b =$ "a", $a' =$ "[a", $b' =$ "a]][a", et supposant que q est très petit, on a $g(a,b) = g(a',b') =$ "[[a][a]][a]". Pour voir que cette situation est bien conflictuelle, considérons les correspondances suivantes établies par la Définition 3.19. Les ψ-programmes a' et b' ne sont sûrement pas dans **range**(g), et correspondraient donc à la fonction indéfinie partout. Donc, $g(a',b')$ devrait correspondre lui aussi à la fonction indéfinie partout. D'autre part, on peut vérifier que a et b correspondent chacun à une fonction totale. Donc, $g(a,b)$ devrait correspondre à une fonction totale ; mais ceci est une contradiction puisque $g(a,b) = g(a',b')$.

Ce problème, cependant, est facile à solutionner. Nous dirons qu'un ψ-programme a est *bien formé* ssi il a le même nombre d'occurrences de "[" et de "]", et aucun préfixe de a n'a plus d'occurrences de "]" que de "[". Nous dirons que a est *malformé* ssi il n'est pas bien formé. On montre par un raisonnement immédiat que la fonction g telle que définie ci-dessus est injective si on la restreint à l'ensemble des programmes bien formés. De plus, la tâche de vérifier si un programme est bien formé ne demande pas plus de temps qu'il n'en faut pour parcourir l'entrée. Ainsi, un algorithme calculant g telle que définie ci-dessus pourrait être modifié de façon à ce qu'il vérifie si ses arguments sont bien formés alors même qu'il les copie des rubans d'entrée au ruban de sortie, et à ce qu'il produise systématiquement un programme malformé si l'un ou l'autre de ses arguments est malformé. Ceci nous amène à notre définition finale de g.

$$g(a,b) \stackrel{\mathrm{d}}{=} \begin{cases} [a][b]\underbrace{*\cdots\cdots\cdots\cdots\cdots*}_{\lfloor (q-1)|a,b| \rfloor \text{ fois}} & \text{si } a \text{ et } b \text{ sont bien formés,} \\ [a][b[\underbrace{*\cdots\cdots\cdots\cdots\cdots*}_{\lfloor (q-1)|a,b| \rfloor \text{ fois}} & \text{autrement.} \end{cases}$$

Clairement, g satisfait encore "l'exigence d'une sortie de longueur qn", et ne cause toujours pas de circularité dans la Définition 3.19. Elle n'est toujours pas injective, mais cela ne cause plus de conflit dans la définition de ψ, puisque nous avons maintenant

$$(\forall a, b, a', b')[g(a,b) = g(a',b') \Longrightarrow \psi_a \circ \psi_b = \psi_{a'} \circ \psi_{b'}],$$

étant donné que tout ψ-programme malformé calcule la fonction indéfinie partout.

On peut montrer que g (dans cette version comme dans la première) est calculable en temps $\lfloor qn \rfloor + k$ pour un certain k petit ; en fait exactement dans le temps requis pour écrire la sortie. Bien que cet état de choses puisse surprendre au premier abord, il est une conséquence du fait que $\lambda n.\lfloor qn \rfloor + 1$ est pleinement constructible en temps (*"fully time-constructible"* [HU79]) pour tout rationnel $q \geq 1$ [Marc], et du lemme suivant.

3.20 Lemme *Pour toute fonction f pleinement constructible en temps, il existe une machine de Turing qui, sur toute entrée, s'arrête en exactement $f(n)$ étapes, où n est la longueur de l'entrée, après avoir parcouru son entrée dans les n premières étapes.*

Preuve. Comme f est pleinement constructible en temps, il existe M, une machine de Turing qui, sur toute entrée, s'arrête en exactement $f(n)$ étapes, où n est la longueur de l'entrée. Nous construisons M', une autre machine de Turing, satisfaisant l'énoncé du lemme. Les seules différences entre M' et M sont que M' a deux rubans de travail de plus que M, un alphabet de ruban légèrement plus grand et, bien sûr, une fonction de transition différente. Nous utilisons sans présentation différentes techniques standard de construction de machines de Turing (voir § 1.3.4).

Supposons que 0 soit un symbole non-blanc de l'alphabet d'entrée de M. Les mouvements de la tête d'entrée de M' sont définis de telle sorte que, peu importe l'entrée, celle-ci est parcourue dans les n premières étapes de calcul, où n est la longueur de l'entrée. Après n étapes, la tête d'entrée de M' reste immobile sur la première case blanche suivant l'entrée, jusqu'à ce que M' s'arrête. Également, à partir de la première étape de calcul, et donc concouramment au parcours de l'entrée (et peut-être aussi subséquemment), M' *simule* M (i.e., imite son comportement) sur entrée 0^n. À la fin de cette

simulation, i.e., après exactement $f(n)$ étapes, M' s'arrête ; elle répond donc aux conditions dans l'énoncé du lemme.

L'entrée 0^n sur laquelle M est simulée n'est *pas* écrite explicitement sur un ruban. Cependant, la simulation est effectuée de telle façon que *conceptuellement*, M traite l'entrée 0^n. Le rôle de chacun des rubans de travail de M (i.e., tous ses rubans sauf celui d'entrée) est joué directement par un ruban de travail de M'. Le rôle du ruban d'entrée de M est joué par un des rubans de travail additionnels de M', appelé *quasi*-ruban d'entrée. Tout ce qui se rapporte au ruban d'entrée de M (mouvements de tête, écriture de symbole, lecture de symbole) est normalement exécuté, dans la simulation, sur le quasi-ruban d'entrée. Cependant, lorsqu'une case de celui-ci est visitée pour la première fois, alors, le symbole considéré lu par la tête d'entrée de M pour les fins de la simulation n'est *pas* celui présentement lu sur le quasi-ruban d'entrée, mais est assumé être 0, si le symbole présentement lu sur le (vrai) ruban d'entrée de M' est non-blanc *ou* si un certain compteur, le *compteur de retard*, est non-nul ; ou est assumé être blanc autrement.

Le *compteur de retard*, un compteur à valeur entière non-négative, est implanté sur le deuxième ruban de travail additionnel de M'. Il donne en tout temps la différence entre le nombre de cases de l'entrée (la vraie) ayant été parcourues par M' jusqu'à présent et le nombre de cases ayant jamais été visitées par M' sur le quasi-ruban d'entrée jusqu'à présent, ou 0 si cette différence est négative. (Le second terme de la différence peut aussi être vu comme étant le nombre de cases ayant jamais été visitées conceptuellement par M sur son "ruban d'entrée" dans la simulation jusqu'à présent.) La valeur du compteur est donnée par la *position* de la tête de lecture sur le ruban. La première case du ruban est "marquée" à la première étape de calcul, de façon à pouvoir détecter une valeur zéro. Le compteur est augmenté (diminué) par déplacement de la tête à droite (à gauche). Le compteur est à zéro ssi la case présentement lue est marquée.

Le lecteur peut d'ores et déjà vérifier que, si le compteur de retard contient bien en tout temps la valeur qu'il est censé contenir, alors la simulation de M s'effectue bien conceptuellement sur entrée 0^n. Pour que ce soit effectivement le cas, il suffit de définir la fonction de transition de M' de telle sorte que le compteur de retard soit augmenté de 1 à chaque étape au début de laquelle la tête du quasi-ruban d'entrée ne visite *pas* une case pour la première fois *et* la tête du (vrai) ruban d'entrée ne lit *pas* un symbole blanc, qu'il soit diminué de 1 à chaque étape au début de laquelle la tête du quasi-ruban d'entrée

visite une case pour la première fois *et* la tête du (vrai) ruban d'entrée lit un symbole blanc *et* le compteur de retard est non-nul, et qu'il soit laissé inchangé dans tous les autres cas. □ **Lemme 3.20**

Notre définition de ψ est maintenant terminée.[4] Il s'agit d'un SP acceptable possédant une instance de comp calculable en temps $\lfloor qn \rfloor + k$ pour un certain k petit.

Il nous reste à montrer que toute instance de s-1-1 pour ψ exige un temps de calcul dans $\Omega(n^{1+\lg q})$. Nous montrons en fait que pour toute instance s de s-1-1 pour ψ, il existe une constante c_0 telle qu'il existe une infinité de p pour lesquels il existe une infinité de x pour lesquels $|s(p,x)| \geq c_0 \cdot |p,x|^{1+\lg q}$. (En fait, nous exhiberons une constante unique c_0, indépendante de s.)

Nous disons qu'un ψ-programme est *atomique* ssi il n'est pas dans $\mathbf{image}(g)$. Il est implicite dans la précédente discussion de g que tout ψ-programme bien formé p peut être "analysé syntaxiquement", i.e., découpé en une séquence non-vide de programmes atomiques (forcément bien formés) qui, lorsque composés suivant *un certain* patron d'application de g, donnent p. (Cette analyse peut être réalisée en suivant une procédure effective, car g est strictement croissante en chaque argument, cependant, nous n'utilisons pas ce fait dans ce qui suit.) Pour tout ψ-programme bien formé p, nous appelons la séquence de programmes atomiques obtenus de p par "analyse syntaxique" la *séquence atomique* de p, et la dénotons par $as(p)$. La longueur de $as(p)$, dénotée $|as(p)|$, est le nombre d'occurrences de programmes qu'elle contient. On note que par définition de ψ et par associativité de la composition de fonctions, la sémantique de tout ψ-programme bien formé dépend uniquement de sa séquence atomique. En d'autres mots, deux ψ-programmes bien formés qui ont la même séquence atomique calculent nécessairement la même fonction partielle.

On se rappelle que tout ψ-programme malformé calcule la fonction indéfinie partout. Un moment de réflexion nous fait réaliser que la même chose est vraie de tout programme bien formé dont la séquence atomique ne contient pas exclusivement des occurrences des programmes `a`, `b` et `c`. Donc, tout ψ-programme calculant autre chose que la fonction indéfinie partout doit nécessairement être bien formé *et* avoir une séquence atomique composée exclusivement d'occurrences des programmes `a`, `b` et `c`. Nous dirons qu'un ψ-programme bien formé est *sensé* ssi sa séquence atomique est composée

4. ψ est très près des *"completions of finite bases"* de Royer [Roy87, Définition 2.4.1].

exclusivement d'occurrences des programmes `a, b` et `c` ; par extension, nous dirons que la séquence atomique elle-même d'un tel programme est sensée.

Il est bien connu que tout SP acceptable a une infinité de programmes pour chaque fonction partielle récursive. Ainsi, il y a une infinité de ψ-programmes calculant la première fonction de projection de $\langle \cdot, \cdot \rangle$, i.e., la fonction $\lambda \langle x, z \rangle .x$. Soit p_0 n'importe lequel de ces programmes, et soit s n'importe quelle instance de `s-1-1` en ψ. Notons tout de suite que pour tout k, le ψ-programme $s(p_0, k)$ calcule la fonction $\lambda z.k$, et doit donc être sensé.

Imaginons que nous calculons successivement $s(p_0, k)$ pour $k = 0,\ 1,\ 2, \ldots$ Avec ces programmes, nous construisons deux listes, que nous appelons SA et $LONMAX$, signifiant respectivement, "séquence atomique" et "longueur maximum". Chaque liste est indexée par k. Pour chaque nouveau ψ-programme $s(p_0, k)$ obtenu, nous remplissons la position k dans chaque liste. Tout d'abord, nous écrivons la séquence atomique de $s(p_0, k)$ sur la liste SA, ensuite, nous écrivons sur la liste $LONMAX$ la longueur de la plus longue séquence atomique figurant à ce moment sur la liste SA.

Tous les ψ-programmes $s(p_0, k)$ sont inéquivalents deux-à-deux, et donc, la liste SA ne peut contenir deux fois la même séquence atomique. D'autre part, toutes les séquences atomiques sur cette liste doivent être sensées, puisqu'aucun des programmes $s(p_0, k)$ ne calcule la fonction indéfinie partout. La liste $LONMAX$ doit donc croître suffisamment rapidement, comme fonction de k, pour qu'il y ait au moins k séquences atomiques sensées distinctes de longueur n'excédant pas $LONMAX(k)$, et ce, pour tout k. Formellement, l'inégalité suivante doit être respectée pour tout k :

$$\sum_{i=1}^{LONMAX(k)} 3^i \geq k. \tag{3.9}$$

De (3.9), on déduit que $LONMAX(k) \geq \log_3(2k/3 + 1)$ pour tout k. Maintenant, $\log_3(2k/3 + 1) \geq |k|/4$ pour tout k suffisamment grand. Manifestement, $LONMAX(k)$ tend vers l'infini avec k, et chaque fois que $LONMAX(k) < LONMAX(k+1)$, on a $|as(s(p_0, k+1))| = LONMAX(k+1)$. Donc, $|as(s(p_0, k))| = LONMAX(k) \geq |k|/4$ pour une infinité de valeurs de k. Le lemme qui suit nous permet de borner inférieurement $|s(p_0, k)|$ pour chacune de ces valeurs de k.

3.21 Lemme *Soit p un ψ-programme sensé. Alors, $|p| \geq mq^{\lfloor \lg m \rfloor}$, où $m = |as(p)|$.*

Preuve. Un *arbre d'analyse* d'un ψ-programme sensé p est une arborescence binaire étiquetée dans laquelle la racine porte l'étiquette p, et dans laquelle chaque noeud est soit étiqueté par un programme atomique et n'a aucun fils, soit étiqueté par $g(a, b)$ pour certains programmes a et b, et a un fils gauche étiqueté a et un fils droit étiqueté b. On montre facilement que tout p sensé a un arbre d'analyse unique, fini, et dont la séquence des feuilles, de gauche à droite, coïncide avec la séquence atomique (et donc, n'a d'occurrences que des programmes atomiques `a`, `b` et `c`).

Soit p un ψ-programme sensé. Soit T l'arbre d'analyse de p, et soit $m \overset{\mathrm{d}}{=} |as(p)|$ (T a donc exactement m feuilles). Définissons w, une fonction associant à chaque noeud de T un *poids* (un nombre réel), comme suit : si v est une feuille, alors $w(v) = 1$; autrement, $w(v) = q(w(\textit{fils-gauche}(v)) + w(\textit{fils-droit}(v)))$. On remarque que, par "l'exigence d'une sortie de longueur qn" satisfaite par g, la longueur de l'étiquette d'un noeud de T est au moins aussi grande que son poids. On définit le poids d'un arbre d'analyse comme étant le poids de sa racine. Nous montrons que le poids de T est au moins $mq^{\lfloor \lg m \rfloor}$, établissant du même coup le lemme.

Le *niveau* d'un noeud dans un arbre d'analyse est sa distance de la racine. La *profondeur* d'un arbre d'analyse est le niveau de sa feuille la plus éloignée de la racine. Il est facile de montrer (par induction sur la profondeur) que le poids d'un arbre d'analyse est donné par la sommation, sur toutes ses feuilles v, de $q^{\ell(v)}$, où $\ell(v)$ est le niveau de v.

Soit T' un arbre d'analyse de poids minimal parmi les arbres d'analyses de m feuilles. Clairement, $w(T) \geq w(T')$. Soit d la profondeur de T'. On note que T' possède au moins deux feuilles de niveau d. Supposons que T' possède une feuille d'un certain niveau $e < d - 1$. En déplaçant deux feuilles du niveau d au niveau $e + 1$, on obtient un nouvel arbre d'analyse, ayant toujours m feuilles, mais dont le poids diffère de celui de T' par la valeur $-2q^d + q^{d-1} + 2q^{e+1} - q^e$. Cette valeur peut être récrite comme $(2 - 1/q)(q^{e+1} - q^d)$, et est donc strictement négative, puisque $e + 1 < d$ et $q > 1$. Comme T' est un arbre d'analyse de poids minimal parmi ceux ayant m feuilles, il faut conclure que toutes ses feuilles sont situées au niveau d ou $d - 1$. Comme T' est une arborescence binaire, $d \geq \lceil \lg m \rceil$. Si m n'est pas une puissance de 2, alors $d - 1 \geq \lfloor \lg m \rfloor$, et donc, $w(T') \geq mq^{\lfloor \lg m \rfloor}$. Si m est une puissance de

2, $d = \lg m = \lfloor \lg m \rfloor$, mais alors *toutes* les feuilles de T' sont au niveau d, et donc, $w(T') \geq mq^{\lfloor \lg m \rfloor}$. □ **Lemme 3.21**

Donc, pour l'infinité de valeurs de k telles que $|as(s(p_0, k))| \geq |k|/4$, on obtient

$$|s(p_0, k)| \geq (|k|/4)q^{\lfloor \lg(|k|/4) \rfloor}.$$

Dès que $k \geq p_0$, on a $|k| \geq |p_0, k|/2$, ce qui nous donne

$$|s(p_0, k)| \geq \frac{1}{8q^4} \cdot |p_0, k|^{1+\lg q}$$

pour une infinité de valeurs de k. □ **Théorème 3.18**

Notons le fait particulier suivant à propos du SP ψ construit dans la preuve précédente. L'instance de s-1-1 la plus naturelle suggérée par la définition de ψ est celle obtenue via la construction de Machtey et Young. Une analyse rapide de l'algorithme résultant de cette construction nous montre qu'il prend un temps double exponentiel. Cependant, on sait par le Théorème 3.13 qu'il *existe* un algorithme pour une instance de s-1-1 en ψ fonctionnant en temps $O(n^{1+\lg q})$. L'aspect particulier de la situation est que, pour *constructivement exhiber* l'algorithme efficace, il semble n'y avoir aucune autre possibilité que de commencer par appliquer la construction de Machtey et Young pour obtenir (par traduction de ϕ-programmes) un ensemble de programmes de base satisfaisant les équations (3.6) de la preuve du Théorème 3.13.

Les Théorèmes 3.13 et 3.18 donnent le corollaire suivant :

3.22 Corollaire *Pour tout rationnel $q > 1$, il existe un SP acceptable possédant une instance de* comp *calculable en temps $\lfloor qn \rfloor + k$ pour un certain k petit, mais n'en possédant aucune calculable en temps $rn + k'$, pour tout $r < q$ et k'.*

Preuve. Soit $q > 1$ un rationnel. Par le Théorème 3.18, il existe ψ, un SP acceptable possédant une instance de comp calculable en temps $\lfloor qn \rfloor + k$ pour un certain k petit, mais dont toutes les instances de s-1-1 requièrent un temps de calcul dans $\Omega(n^{1+\lg q})$. Si ψ possédait une instance de comp calculable en temps $rn + k'$ pour un certain $r < q$ (prenons, sans perte de généralité, $r > 1$) et un certain k', alors par le Théorème 3.13, il posséderait

une instance de s-1-1 calculable en temps $O(n^{1+\lg r})$, une contradiction. □

La construction dans notre preuve du Théorème 3.18 est applicable à n'importe quelle fonction pleinement constructible en temps. Comme dans le cas du Théorème 3.13, un résultat vraiment général semble difficile à énoncer sans l'introduction d'une opération *ad hoc* sur les classes de fonctions. Cependant, nous avons le corollaire suivant :

3.23 Corollaire *Il existe un SP acceptable possédant une instance de* comp *calculable en temps polynomial, mais dont toutes les instances de* s-1-1 *requièrent un temps de calcul dans* $2^{\Omega(n)}$.

Esquisse de preuve. On modifie la preuve du théorème comme suit. On choisit une g qui "rembourre" sa sortie de telle façon que $|g(a,b)| \geq |a,b|^2$ pour tout a et b. Aussi, on substitue aux programmes `a`, `b` et `c` dans la Définition 3.19 les programmes `aa`, `bb` et `cc`. Ainsi, tout programme survenant dans une séquence atomique sensée a longueur 2. Puis, en remplacement du Lemme 3.21, l'argument suivant montre que pour tout programme sensé p, $|p| \geq 2^{2^{\lceil \lg m \rceil}}$, où $m = |as(p)|$. Considérons l'arbre d'analyse de p, défini comme dans le Lemme 3.21, mais dans lequel les poids sont attribués comme suit : une seule feuille, parmi celles qui sont les plus éloignées de la racine, reçoit poids 2 ; toutes les autres feuilles reçoivent poids 0 ; chaque noeud interne reçoit un poids égal au carré de la somme des poids de ses fils. Clairement, le poids de cet arbre est au moins $2^{2^{\lceil \lg m \rceil}}$, et constitue une borne inférieure sur la longueur de p. À partir de là, le raisonnement est comme dans la preuve du théorème. □

Le prochain corollaire découle immédiatement du précédent et du Théorème 3.13.

3.24 Corollaire *Il existe un SP acceptable possédant une instance de* s-1-1 *dans* $\mathcal{P}$*time, mais n'en possédant aucune dans* $\mathcal{L}$*intime.*

3.3.3 Remarques concernant s-1-1 contre comp

Tous les résultats de cette section ont été établis avec $|a| + |b|$ comme définition sous-jacente de la longueur de deux arguments a et b. Il est intéressant

de voir jusqu'à quel point ces résultats sont dépendants de cette définition. En fait, on peut montrer que si on utilise $|\langle a, b\rangle|$ comme définition de la longueur de deux arguments a et b, alors on obtient une borne de $n^{2+\lg q}$, et pour le Théorème 3.13 et pour le Théorème 3.18 (de plus, il est possible d'utiliser une fonction g *injective* dans la preuve de ce dernier). Si on utilise $\max(|a|, |b|)$ comme définition de la longueur de deux arguments a et b, alors on obtient $n^{\lg q}$ (si $q > 2$) ou $n \log n$ (si $q = 2$) comme borne dans les deux théorèmes. Donc, avec les trois définitions, les bornes supérieures et inférieures coïncident.

Comme le montre la prochaine proposition, la restriction $q \geq 2$ qu'on a avec la troisième définition n'en est pas vraiment une.

3.25 Proposition *Supposons que c soit une instance (pas nécessairement effective) de* comp *dans un SP maximal quelconque (pas nécessairement exécutable). Alors, il n'existe aucun $r < 2$ tel que pour un certain k fixe et pour tout i et j, $|c(i, j)| \leq r \cdot \max(|i|, |j|) + k$.*

Preuve. Supposons qu'il existe un $r < 2$ et un k tels que pour tout i et j, $|c(i, j)| \leq r \cdot \max(|i|, |j|) + k$. Soient p_0 et p_1 des programmes (dans le SP pour lequel c est une instance de comp) pour, respectivement, $\lambda z.2z$ et $\lambda z.2z + 1$. Définissons $P_0 \stackrel{\mathrm{d}}{=} \{p_0, p_1\}$ et recursivement, pour $n \geq 1$, $P_n \stackrel{\mathrm{d}}{=} \{c(p, q) \mid p, q \in P_{n-1}\}$. Soit n fixé. Il est facile de voir que pour toute chaîne de bits de longueur 2^n, il existe dans P_n un programme dont l'action sur toute entrée non-nulle est de lui concaténer la chaîne de bits en question. Comme il y a 2^{2^n} chaînes de bits distinctes de longueur 2^n, il doit y avoir au moins 2^{2^n} programmes distincts dans P_n.

Posons maintenant $\ell_0 \stackrel{\mathrm{d}}{=} \max(|p_0|, |p_1|)$ et récursivement, pour $n \geq 1$, $\ell_n = r\ell_{n-1} + k$. Par notre hypothèse sur c, il est clair que pour tout n, pour tout $p \in P_n$, $|p| \leq \ell_n$. Si $r \neq 1$, alors $\ell_n \leq (\ell_0 + k/(r-1))r^n$, autrement, $\ell_n = \ell_0 + kn$. Dans tous les cas, pour toute valeur suffisamment grande de n, $\ell_n < 2^n$. Ceci est une contradiction puisque P_n contient au moins 2^{2^n} programmes distincts et qu'il n'y a que $2^{2^n - 1}$ programmes de longueur inférieure à 2^n. □

La proposition précédente constitue une borne inférieure absolue à laquelle est sujette *toute* instance de comp (effective ou non) dans n'importe quel SP de puissance de calcul maximale (exécutable ou non). Notez cependant que

nous n'avons pas éliminé la possibilité d'une instance de comp c (effective ou non) pour laquelle $\lim_{n\to\infty}[\inf_{i,j>n}(2\cdot\max(|i|,|j|)-|c(i,j)|)]=\infty$, ou même $\lim_{n\to\infty}[\inf_{i+j>n}(2\cdot\max(|i|,|j|)-|c(i,j)|)]=\infty$. Nous soupçonnons qu'il existe de telles instances *effectives* dans certains SP *acceptables*.

Le fait que nous obtenions un logarithme en base 2 dans nos bornes des Théorèmes 3.13 et 3.18 vient du fait que comp correspond à la composition de deux programmes. Si nous utilisions une autre RS correspondant à la composition de $m>2$ programmes, nous obtiendrions un logarithme en base m. Nous obtiendrions aussi une borne inférieure de m dans la Proposition 3.25.

Nous mentionnons le fait que, dans notre preuve du Théorème 3.13, nous aurions pu n'utiliser que trois programmes de base, qui auraient alors calculé les trois fonctions $\lambda z.\langle 0,z\rangle$, $\lambda\langle y,z\rangle.\langle 2y,z\rangle$ et $\lambda\langle y,z\rangle.\langle 2y+1,z\rangle$. Nous aurions alors aussi traité l'entier donné en entrée (d) en commençant par le bit le *moins* significatif.

Le fait suivant mérite d'être mentionné. Dans notre preuve du Théorème 3.18, nous bornons inférieurement la sortie (i.e., la valeur) de *toutes* les instances de $\mathsf{s\text{-}1\text{-}1}$ pour ψ, pas seulement les instances *effectives*. Nous avons donc le corollaire suivant à la preuve du Théorème 3.18 (nous n'énonçons que le cas $q>1$) :

3.26 Corollaire *Pour tout rationnel $q>1$, il existe un SP acceptable ψ possédant une instance de* comp *calculable en temps $qn+k$ pour un certain k, mais tel que pour* toute *instance s de* $\mathsf{s\text{-}1\text{-}1}$ *en ψ, $|s(p,x)|\in\Omega(|p,x|^{1+\lg q})$.*

Maintenant, on note aussi que, dans notre preuve du Théorème 3.13, si l'instance de comp de départ c, sans être *calculable* en temps $qn+k$, satisfait $|c(p_1,p_2)|\leq q\cdot|p_1,p_2|+k$ pour un certain k, alors nous obtenons une instance de $\mathsf{s\text{-}1\text{-}1}$ qui, sans être nécessairement *calculable* en temps $O(n^{1+\lg q})$, satisfait quand même $|s(p,d)|\in O(|p,d|^{1+\lg q})$. Le corollaire précédent donne donc l'analogue suivant du Corollaire 3.22 :

3.27 Corollaire *Pour tout rationnel $q>1$, il existe un SP acceptable ψ possédant une instance de* comp *calculable en temps $\lfloor qn\rfloor+k$, mais tel que pour* toute *instance c de* comp *en ψ, pour tout $r>q$ et tout k', $|c(p_1,p_2)|\geq r\cdot|p_1,p_2|+k'$ infiniment souvent.*

Les Corollaires 3.23 et 3.24 peuvent aussi être généralisés de la sorte.

Il est cependant intéressant de noter qu'il existe des SP acceptables qui témoignent de la véracité du Théorème 3.18, mais *non* de la véracité du Corollaire 3.26. Plus précisément (nous n'énonçons encore ici que le cas $q > 1$) :

3.28 Théorème *Pour tout rationnel $q > 1$, il existe un SP acceptable possédant une instance de* `comp` *calculable en temps $qn + k$ pour un certain k, dont toutes les instances effectives de* `s-1-1` *requièrent un temps de calcul dans $\Omega(n^{1+\lg q})$, mais pour lequel il existe une instance (non-effective) s de* `s-1-1` *satisfaisant $|s(p,x)| \in O(|p,x|)$.*

Esquisse de preuve. La preuve est une modification de celle du Théorème 3.18. Nous utilisons l'existence (démontrée ci-dessous) d'un SP exécutable η tel que $A \stackrel{\mathrm{d}}{=} \{p \mid \mathbf{image}(\eta_p) \text{ est infinie}\}$ est immunisé, et tel que pour tout i, il existe un $j \leq i$ pour lequel $\eta(\langle i,j\rangle) = \phi_i$ (η a donc puissance de calcul maximale).

Soit $\{\mathtt{a},\mathtt{b}\}^+$ l'ensemble des chaînes finies non-vides sur l'alphabet $\{\mathtt{a},\mathtt{b}\}$. Pour tout $a \in \{\mathtt{a},\mathtt{b}\}^+$, nous dénotons par $\ell(a)$ la position de a dans $\{\mathtt{a},\mathtt{b}\}^+$ selon l'ordre lexicographique usuel (avec $\mathtt{a}$ occupant la position 0). On peut regarder ℓ comme une bijection entre $\{\mathtt{a},\mathtt{b}\}^+$ et N, et il est clair que pour tout $a \in \{\mathtt{a},\mathtt{b}\}^+$ et tout $n \in N$, on peut trouver respectivement $\ell(a)$ et $\ell^{-1}(n)$ en temps linéaire.

La fonction g est comme dans la preuve du Théorème 3.18. Nous définissons ψ ainsi :

3.29 Définition

$$\begin{array}{lcll} \psi(a) & \stackrel{\mathrm{d}}{=} & \eta_{\ell(a)}, & \text{pour tout } a \in \{\mathtt{a},\mathtt{b}\}^+, \\ \psi(g(a,b)) & \stackrel{\mathrm{d}}{=} & \psi_a \circ \psi_b, & \text{pour tout } a, b, \\ \psi(p) & \stackrel{\mathrm{d}}{=} & \lambda z.\uparrow & \text{pour tout } p \notin \{\mathtt{a},\mathtt{b}\}^+ \cup \mathbf{image}(g). \end{array}$$

La fonction g est encore une instance de `comp` pour ψ calculable en temps $qn+k$ pour un certain k petit. Comme η est de puissance de calcul maximale, ψ l'est aussi et donc, par le Théorème 2.69, il est acceptable.

Les notions de ψ-programme bien formé et de séquence atomique d'un ψ-programme bien formé sont définies comme dans la preuve du Théorème 3.18.

Dans la phase de construction des deux listes SA et $LONMAX$, nous utilisons un ψ-programme p_0 calculant la fonction d'addition $(\lambda\langle x, y\rangle.x + y)$, et une instance *effective* s de s-1-1 pour ψ. Alors, les programmes $s(p_0, k)$, $k = 0,\ 1,\ 2, \ldots$, non seulement sont inéquivalents deux-à-deux, mais encore, calculent tous des fonctions d'image infinie. Notons qu'alors, tous les programmes atomiques figurant sur la liste SA *doivent* être de la forme $\ell^{-1}(n)$ pour un certain $n \in A$: en effet, si un programme bien formé possède dans sa séquence atomique un programme d'une autre forme, il calcule nécessairement un fonction d'image finie. Comme s est récursive, et A, immunisé, on déduit qu'un nombre *fini* de programmes atomiques distincts figurent sur la liste SA ; appelons ce nombre fini b.

Comme analogue à l'inégalité (3.9 ; p. 107), on a alors

$$\sum_{i=1}^{LONMAX(k)} b^i \geq k$$

pour tout k. La suite du raisonnement est comme dans la preuve du Théorème 3.18 ; notons cependant que la constante multiplicative que l'on obtient pour borner inférieurement $|s(p_0, k)|$ par rapport à $|p_0, k|^{1+\lg q}$ *dépend* maintenant de b, et donc de l'instance s de s-1-1 considérée.

Nous argumentons maintenant que ψ possède une instance (non-effective) s de s-1-1 satisfaisant $|s(p, x)| \in O(|p, x|)$.

On note d'abord que comme pour tout i, il existe un $j \leq i$ tel que $\eta(\langle i, j\rangle) = \phi_i$, alors il existe une fonction t (non-récursive) telle que pour tout i, $\eta_{t(i)} = \phi_i$ et telle que, de plus, $|t(i)| \in O(|i|)$. Si on définit $t' \stackrel{\mathrm{d}}{=} \ell^{-1} \circ t$, on a alors que pour tout i, $\psi_{t'(i)} = \phi_i$ et que, de plus, $|t'(i)| \in O(|i|)$.

Soient s_ϕ une instance dans $\mathcal{L}intime$ de s-1-1 pour ϕ et i_0 un ϕ-programme pour $\widehat{\psi}$ (autrement dit, un interprète pour ψ). On note que pour tout p, $\phi(s_\phi(i_0, p)) = \psi_p$. On vérifie alors facilement que $s \stackrel{\mathrm{d}}{=} \lambda p, x.[t'(s_\phi(s_\phi(i_0, p), x))]$ est une instance de s-1-1 pour ψ. De plus, par le fait que s_ϕ est calculable en temps linéaire et que $|t'(i)| \in O(|i|)$, on montre facilement que $|s(p, x)| \in O(|p, x|)$. $\square$

Le lemme suivant établit l'existence du SP η utilisé dans la preuve précé-

dente.

3.30 Lemme *Il existe un SP exécutable η tel que $\{p \mid$ **image**(η_p) est infinie$\}$ est immunisé et tel que pour tout i, il existe un $j \leq i$ pour lequel $\eta(\langle i, j \rangle) = \phi_i$.*

Preuve. Nous commençons par modifier légèrement la construction de Post d'un ensemble simple [Rog87] de façon à obtenir un ensemble simple S doté de la propriété que pour tout i, il existe un $j \leq i$ tel que $\langle i, j \rangle \notin S$. Nous construisons S comme suit :

Par queue de colombe, nous énumérons W_i pour tout $i \in N$. Pour chaque i, nous plaçons dans S le premier membre x obtenu dans l'énumération de W_i qui satisfait $x > \langle i, i \rangle$, si un tel membre est jamais obtenu. On dit alors que *i cause l'entrée* de x dans S (l'entrée d'un x donné peut être causée par plus d'un i). Clairement, S est r.é. Notons que $\lambda z. \langle z, z \rangle$ est strictement croissante et qu'en conséquence, pour tout i et tout $j > i$, si j cause l'entrée d'un membre dans S, ce membre est sûrement supérieur à $\langle i, i \rangle$. Donc pour tout i, au plus i des éléments de $\{0,\ 1, \ldots,\ \langle i, i \rangle\}$ sont dans S, et il existe forcément un $j \leq i$ tel que $\langle i, j \rangle \notin S$. Ceci, à son tour, implique (par injectivité de $\langle \cdot, \cdot \rangle$) que $\overline{S}$ est infini. Clairement, tout i tel que W_i est infini cause l'entrée d'un x dans S, et donc, a intersection non-vide avec S. Le complément de S répond à la définition d'ensemble immunisé, et comme S est r.é., S est simple.

Nous définissons maintenant η en donnant un algorithme pour interpréter les η-programmes, qui sont vus comme des *paires*.

> **Entrée :** $\langle \langle a, b \rangle, z \rangle$
> $\{$ $\langle a, b \rangle$ est un η-programme et z est une donnée $\}$
> **Algorithme pour $\widehat{\eta}$:**
> 1. Effectuer z étapes dans l'énumération des membres de S ; si $\langle a, b \rangle$ est obtenu au cours de l'opération, retourner 0.
> 2. Retourner $\phi_a(z)$.
>
> $\Box$ **Algorithme**

Il est clair que pour tout η-programme $\langle a, b \rangle \notin S$, $\eta(\langle a, b \rangle) = \phi_a$. Comme pour tout i, il existe un j tel que $\langle i, j \rangle \notin S$, on déduit que η a puissance de calcul maximale.

D'autre part, tout η-programme $\langle a, b \rangle \in S$ est obtenu à l'Étape 1 après un nombre fini d'étapes dans l'énumération de S, et calcule donc un variante

finie de $\lambda z.0$. Il s'ensuit que l'ensemble $A \stackrel{\mathrm{d}}{=} \{p \mid \mathbf{image}(\eta_p) \text{ est infinie}\}$ est inclus dans $\overline{S}$. Comme $\overline{S}$ est immunisé, A ne peut être que fini ou immunisé. Or, η a puissance de calcul maximale, et A ne peut être fini ; il est donc immunisé. □

Notons que l'ensemble A de la preuve précédente est majoré par $\lambda i.\langle i, i\rangle$, et ne peut donc être hyperimmunisé. Le Lemme 3.30 n'est donc pas un cas particulier du Théorème 3.1 de [Roy87] (qui implique l'existence d'un SP exécutable de puissance de calcul maximale dont l'ensemble des programmes calculant des fonctions partielles d'image infinie est *hyper*immunisé).

Nous terminons cette discussion par deux questions ouvertes.

D'abord, une question informelle au sujet des Théorèmes 3.13 et 3.18 est de savoir si ces résultats peuvent être généralisés *élégamment*, sans recourir à des définitions *ad hoc*. Autrement dit, existe-t-il une relation "simple" entre classes de complexité qui corresponde de façon satisfaisante à l'interrelation réelle entre comp et s-1-1 du point de vue de la complexité des instances dans un même SP ?

La deuxième question ouverte concerne la classe des SP acceptables possédant une instance dans $\mathcal{P}time$ de s-1-1. Intuitivement, par le Corollaire 3.5, tout SP dans cette classe possède une "tâche de programmer" qui est "pratiquement réalisable", sous l'hypothèse que les fonctions dans $\mathcal{P}time$ sont celles qui sont "pratiquement calculables". Pour les besoins de la présente discussion, appelons ces SP acceptables les systèmes de programmation *pratiques*. Par le Théorème 3.13, nous avons établi que tout SP exécutable maximal possédant une instance dans $\mathcal{L}intime$ de comp est pratique. Mais l'inverse est-il vrai ? Est-ce que tout SP pratique possède nécessairement une instance dans $\mathcal{L}intime$ de comp ? Nous soupçonnons que la réponse à cette question est négative. Notons au passage qu'il est facile de démontrer que la classe des SP pratiques coïncide avec la classe $GN\mathcal{P}time$ de Hartmanis et Baker [You88, HB75], la classe des SP (ou "Gödel Numberings") exécutables dans lesquels tout autre SP exécutable est traduisible par une fonction dans $\mathcal{P}time$.

Chapitre 4

Relations sémantiques dans les systèmes de programmation maximaux

Dans ce chapitre, nous étudions les interrelations de garantie qui existent entre certaines RS dans le contexte des SP maximaux qui ne sont pas nécessairement exécutables.

Une interrelation de garantie qui est valide dans le contexte des SP maximaux est clairement valide aussi dans le contexte des SP exécutables maximaux. D'autre part, Riccardi [Ric82] a montré que certaines interrelations de garantie valides dans le contexte des SP exécutables maximaux ne sont *pas* valides dans le contexte des SP maximaux. Un intérêt de l'étude des interrelations de garantie dans le contexte général des SP maximaux est qu'elle permet de comparer la puissance expressive de RS qui, dans le contexte des SP *exécutables* maximaux, sont équivalentes. C'est le cas des RS étudiées dans ce chapitre (sauf la Section 4.4) : elles sont toutes équivalentes dans le contexte des SP exécutables maximaux.

On peut dire que les interrelations de garantie valides dans le contexte des SP maximaux révèlent plus sur la puissance purement "combinatoire" des RS concernées que les interrelations dans le contexte des SP exécutables maximaux. En effet, le fait qu'une RS rs-a garantisse une RS rs-b dans les SP maximaux nous dit que dans n'importe quel SP (même les SP *exécutables*)

on peut, à partir d'une instance effective de rs-a et peut-être de certains programmes spécifiques, construire une instance effective de rs-b *sans connaître un interprète du langage* (si la connaissance d'un interprète était nécessaire, l'interrelation de garantie ne pourrait être valide dans les SP maximaux, puisqu'en général, ils ne possèdent *pas* d'interprète). Par contraste, si l'interrelation de garantie n'est valide que dans les SP *exécutables* maximaux, alors dans au moins un SP *exécutable*, toute utilisation de la technique de programmation correspondant à rs-b, basée sur la seule maîtrise de la technique correspondant à rs-a, *exige* la disponibilité d'un interprète du SP. En ce sens, les interrelations valides dans *tous* les SP maximaux sont intuitivement plus "robustes" que celles qui ne sont valides que dans les SP exécutables maximaux.

Nos résultats des Sections 4.1, 4.2 et 4.3 complètent certains résultats de [Ric82]. Dans les deux premières de ces sections, nous étudions s-1-1 et comp en relation avec une forme affaiblie de comp ; dans la troisième, nous étudions toutes ces RS en relation avec une deuxième forme affaiblie de comp. Nous étudions aussi, dans la Section 4.4, prog en relation avec les RS de Myhill-Shepherdson. Toutes les RS étudiées dans ce chapitre sont aussi des structures de contrôle, en fait, il s'agit de RS *extensionnelles sans récursion au sens de Royer* (Définition 2.33).

Les résultats des Sections 4.1, 4.2 et 4.3 sont présentés schématiquement sur la Figure 4. Chaque région sur cette figure représente la classe des SP maximaux dotés d'une instance effective de la RS identifiant la région. Les lettres grecques réfèrent aux SP utilisés dans nos preuves.

4.1 Pré- et post-compositions hybrides

Nous introduisons d'abord deux variantes de comp, qui s'avéreront strictement plus faibles que comp.

4.1 Définition (a) pr-comp-h $\stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, i, j et q, $\mathcal{R}(\psi, i, j, q) \Leftrightarrow [\psi_q = \psi_i \circ \phi_j]$.

(b) po-comp-h $\stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, i, j et q, $\mathcal{R}(\psi, i, j, q) \Leftrightarrow [\psi_q = \phi_i \circ \psi_j]$.

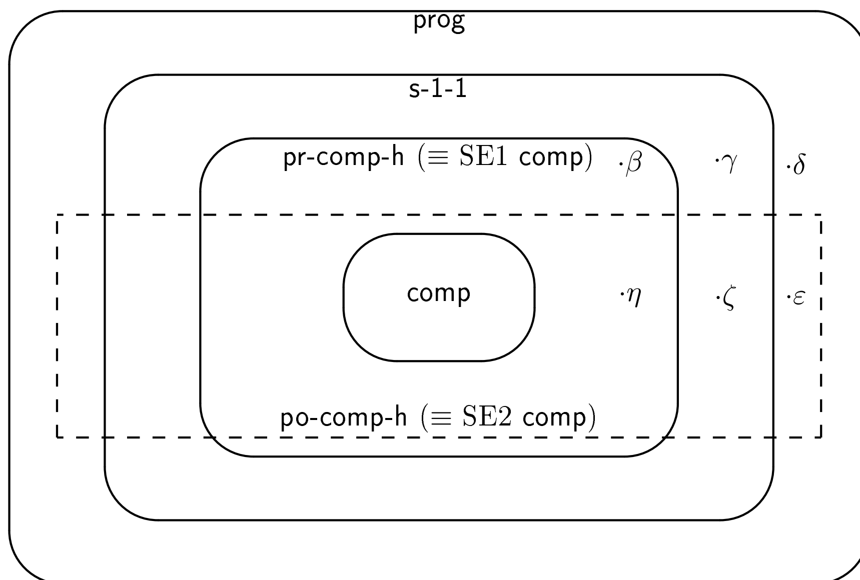

FIGURE 4.1 – Sommaire des résultats des Sections 4.1, 4.2 et 4.3.

Nous appelons la technique de programmation correspondant à $\mathsf{pr\text{-}comp\text{-}h}$ "pré-composition hybride" et celle correspondant à $\mathsf{po\text{-}comp\text{-}h}$, "post-composition hybride". La justification de ces noms est évidente puisqu'une instance de $\mathsf{pr\text{-}comp\text{-}h}$ (respectivement, $\mathsf{po\text{-}comp\text{-}h}$) dans un SP donné permet de pré- (respectivement, post-) composer des programmes d'un SP standard (e.g., des descriptions de machines de Turing) avec les programmes "natifs" du SP considéré.

La définition suivante est simplement la transposition dans le monde des RS de la Définition 1.4.1.2 de [Roy87] :

4.2 Définition (a) Pour toute α partielle récursive, $\mathsf{right\text{-}comp}_\alpha \stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, i et q, $\mathcal{R}(\psi, i, q) \Leftrightarrow [\psi_q = \psi_i \circ \alpha]$.

(b) Pour toute α partielle récursive, $\mathsf{left\text{-}comp}_\alpha \stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, i et q, $\mathcal{R}(\psi, i, q) \Leftrightarrow [\psi_q = \alpha \circ \psi_i]$.

Riccardi a introduit une notion d'instance *semi-effective* d'un schéma de structure de contrôle à deux entrées [Ric80, Définition 1.14]. Riccardi ne s'est intéressé qu'aux instances "semi-effectives en leur second argument" (définition ci-après). Nous donnons le complément logique à la définition de Riccardi :

4.3 Définition (a) Une fonction 2-aire f est dite *semi-effective en son premier argument* (abrégé *SE1*) ssi pour tout y, $\lambda x.f(x, y)$ est récursive.

(b) Une fonction 2-aire f est dite *semi-effective en son second argument* (abrégé *SE2*) ssi pour tout x, $\lambda y.f(x, y)$ est récursive.

Nous notons le fait suivant pour référence future.

4.4 Fait Un SP ψ possède une instance SE1 de comp ssi il possède une instance *effective* de $\mathsf{right\text{-}comp}_\alpha$ pour *toute* $\alpha \in \mathbf{image}(\psi)$. De même, ψ possède une instance SE2 de comp ssi il possède une instance *effective* de $\mathsf{left\text{-}comp}_\alpha$ pour *toute* $\alpha \in \mathbf{image}(\psi)$.

Dans le but de diminuer la longueur des preuves de ce chapitre, nous adoptons les conventions suivantes :

4.5 Conventions 1. c_ϕ dénote une instance effective de comp pour ϕ.

2. s_ϕ dénote une instance effective de s-1-1 pour ϕ.
3. u dénote la fonction universelle unaire de ϕ.
4. id dénote un ϕ-programme pour l'identité.
5. pd dénote un ϕ-programme pour $\lambda z.z + 2$.
6. K dénote l'ensemble r.é. non-récursif de § 1.3.7.

Le fait suivant est peut-être un peu moins immédiat que le précédent :

4.6 Proposition (a) *Un SP maximal possède une instance SE1 de* comp *ssi il possède une instance effective de* pr-comp-h.

(b) *Un SP maximal possède une instance SE2 de* comp *ssi il possède une instance effective de* po-comp-h.

Preuve. Nous ne démontrons que la partie (a) ; la partie (b) se démontre de façon tout à fait similaire (avec $q \stackrel{\text{d}}{=} \lambda i, j.[g_u \circ (g_k)^j \circ g_h(i)]$ pour la direction "$\Rightarrow$").

$\Rightarrow$: La preuve utilise un argument similaire à la construction de Machtey et Young d'une instance de s-1-1 à partir d'une instance de comp [MY78, Théorème 3.1.2]. Soit $\psi \in \mathcal{SPM}$ possédant une instance SE1 de comp. Par le Fait 4.4, ψ possède une instance effective de right-comp$_\alpha$ pour toute $\alpha \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$. Pour chaque $\alpha \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$, soit g_α une instance effective de right-comp$_\alpha$ en ψ.

Soient h, k et q définies comme suit :

$$\begin{aligned} h &\stackrel{\text{d}}{=} \lambda z.\langle 0, z\rangle, \\ k &\stackrel{\text{d}}{=} \lambda\langle y, z\rangle.\langle y+1, z\rangle, \\ q &\stackrel{\text{d}}{=} \lambda i, j.g_h \circ (g_k)^j \circ g_u(i). \end{aligned}$$

Clairement, q est récursive. De plus, par définition de right-comp, pour tout i et j,

$$\begin{aligned} \psi(q(i,j)) &= \psi(g_h \circ (g_k)^j \circ g_u(i))), \\ &= \psi((g_k)^j \circ g_u(i)) \circ h, \\ &= \psi(g_u(i)) \circ k^j \circ h, \\ &= \psi_i \circ u \circ k^j \circ h, \\ &= \psi_i \circ \lambda x.u(\langle j, x\rangle), \\ &= \psi_i \circ \phi_j. \end{aligned}$$

La fonction q est donc une instance effective de $\mathsf{pr\text{-}comp\text{-}h}$ pour ψ.

$\Leftarrow$: Cette direction n'exige pas que le SP ait puissance de calcul maximale. Soient ψ un SP et q une instance effective de $\mathsf{pr\text{-}comp\text{-}h}$ dans ψ. Choisissons $\alpha \in \mathcal{PartRecUn}$ et soit j tel que $\phi_j = \alpha$. Définissons $g \stackrel{\mathrm{d}}{=} \lambda i.q(i,j)$. Par définition de $\mathsf{pr\text{-}comp\text{-}h}$, on a que pour tout i,

$$\psi(g(i)) = \psi(q(i,j)) = \psi_i \circ \phi_j = \psi_i \circ \alpha.$$

Donc, g est une instance effective de $\mathsf{right\text{-}comp}_\alpha$ dans ψ, et donc, ψ possède une instance effective de $\mathsf{right\text{-}comp}_\alpha$ pour toute $\alpha \in \mathcal{PartRecUn}$. Comme forcément $\mathbf{image}(\psi) \subseteq \mathcal{PartRecUn}$, alors, par le Fait 4.4, ψ possède une instance SE1 de comp. $\square$

4.2 pr-comp-h **contre** comp, s-1-1 **et** prog

Nous sommes maintenant prêts pour une première série de résultats. On écrit $\mathsf{rs\text{-}a} \models \mathsf{rs\text{-}b} \models \mathsf{rs\text{-}c}$ ssi $\mathsf{rs\text{-}a} \models \mathsf{rs\text{-}b}$ et $\mathsf{rs\text{-}b} \models \mathsf{rs\text{-}c}$.

4.7 Théorème (a) $\mathsf{comp} \models_m \mathsf{pr\text{-}comp\text{-}h} \models_m \mathsf{s\text{-}1\text{-}1} \models_m \mathsf{prog}$.
(b) $\mathsf{pr\text{-}comp\text{-}h} \not\models_m \mathsf{comp}$.
(c) $\mathsf{s\text{-}1\text{-}1} \not\models_m \mathsf{pr\text{-}comp\text{-}h}$.
(d) $\mathsf{prog} \not\models_m \mathsf{s\text{-}1\text{-}1}$.

Preuve. (a) : Rogers [Rog58] a montré implicitement que $\mathsf{s\text{-}1\text{-}1} \models_m \mathsf{prog}$. Riccardi [Ric80, Théorème 2.6], reprenant essentiellement la preuve de Machtey et Young [MY78, Théorème 3.1.2] a montré que $\mathsf{comp} \models_m \mathsf{s\text{-}1\text{-}1}$. Une instance effective de comp étant bien sûr SE1, il s'ensuit, par la Proposition 4.6 (a), que $\mathsf{comp} \models_m \mathsf{pr\text{-}comp\text{-}h}$. Il reste donc à montrer que $\mathsf{pr\text{-}comp\text{-}h} \models_m \mathsf{s\text{-}1\text{-}1}$.

Soient $\psi \in \mathcal{SPM}$ et q une instance effective de $\mathsf{pr\text{-}comp\text{-}h}$ pour ψ. Soit f une fonction récursive telle que pour tout d, $\phi_{f(d)} = \lambda z.\langle d, z\rangle$. Définissons maintenant $s \stackrel{\mathrm{d}}{=} \lambda p, d.q(p, f(d))$. On a alors pour tout p et d

$$\psi(s(p,d)) = \psi(q(p, f(d))) = \psi_p \circ \phi_{f(d)} = \psi_p \circ \lambda z.\langle d, z\rangle = \lambda z.\psi_p(d,z).$$

On conclut que s est une instance effective de $\mathsf{s\text{-}1\text{-}1}$ pour ψ, et donc que $\mathsf{pr\text{-}comp\text{-}h} \models_m \mathsf{s\text{-}1\text{-}1}$.

Nous exhibons maintenant trois SP, β, γ et δ qui témoignent respectivement de la véracité de (b), (c) et (d).

(b) : Le SP β est défini ainsi :

$$\begin{aligned} \beta(2n) &\stackrel{\mathrm{d}}{=} \phi_n, \\ \beta(2 \cdot \langle a, b \rangle + 1) &\stackrel{\mathrm{d}}{=} \begin{cases} \lambda z.0 \circ \phi_a & \text{si } b \in K, \\ \lambda z.1 \circ \phi_a & \text{autrement.} \end{cases} \end{aligned}$$

Il est clair que $\beta \in \mathcal{SPM}$. Montrons que $\mathsf{pr\text{-}comp\text{-}h} \in \mathsf{UTIL}(\beta)$. Définissons la fonction q par

$$q(i,j) \stackrel{\mathrm{d}}{=} \begin{cases} 2 \cdot c_\phi(i/2, j) & \text{si } i \text{ est pair,} \\ 2 \cdot \langle c_\phi(a,j), b \rangle + 1 & \text{si } i \text{ est impair, où } \langle a, b \rangle = (i-1)/2. \end{cases}$$

On vérifie facilement que q est une instance effective de $\mathsf{pr\text{-}comp\text{-}h}$ pour β, et on conclut donc que $\mathsf{pr\text{-}comp\text{-}h} \in \mathsf{UTIL}(\beta)$. Supposons maintenant qu'il existe c, une instance effective de comp pour β. Soit f la fonction (a priori partielle) calculée par l'algorithme suivant :

> **Entrée :** x
> **Algorithme pour f :**
> 1. $z \leftarrow c(2 \cdot pd, 2 \cdot \langle id, x \rangle + 1)$.
> 2. Retourner 1 si $\phi_{z/2}(0)\downarrow = 2$, et 0 si $\phi_{z/2}(0)\downarrow \neq 2$.
>
> □ **Algorithme**

Nous affirmons que f est récursive et coïncide avec la fonction caractéristique de K. En effet, pour tout x, le z obtenu à l'Étape 1 de l'algorithme est un β-programme calculant une fonction constante, d'image $\{2\}$ si $x \in K$ et $\{3\}$ autrement. Ce z ne peut être impair, car tous les β-programmes impairs calculent des fonctions dont l'image est incluse dans $\{0,1\}$. Donc, par la définition de β, $\beta_z = \phi_{z/2}$, et donc, l'Étape 2 se termine toujours et notre affirmation est vérifée. Elle constitue bien entendu une contradiction (de la non-récursivité de K), de laquelle on conclut que $\mathsf{comp} \notin \mathsf{UTIL}(\beta)$, et donc que $\mathsf{pr\text{-}comp\text{-}h} \not\models_m \mathsf{comp}$.

(c) : On définit γ comme suit :

$$\begin{aligned}\gamma(2n) &\stackrel{\mathrm{d}}{=} \phi_n,\\ \gamma(2n+1) &\stackrel{\mathrm{d}}{=} \begin{cases}\lambda z.1 & \text{si } n \in K,\\ \lambda z.0 & \text{autrement.}\end{cases}\end{aligned}$$

Clairement, $\gamma \in \mathcal{SPM}$. On vérifie aussi sans difficulté que la fonction s définie par

$$s(i,y) \stackrel{\mathrm{d}}{=} \begin{cases} 2 \cdot s_\phi(i/2, y) & \text{si } i \text{ est pair,}\\ i & \text{si } i \text{ est impair}\end{cases}$$

est une instance effective de s-1-1 dans γ. Donc, s-1-1 $\in \mathsf{UTIL}(\gamma)$. Supposons maintenant qu'il existe q, une instance effective de pr-comp-h pour γ. Soit p un ϕ-programme tel que

$$\phi_p = \lambda z. \begin{cases} 0 & \text{si } z = 0,\\ \uparrow & \text{autrement.}\end{cases}$$

Alors, nous affirmons que $\lambda x.\phi_{q(2x+1,p)/2}(0)$ est une fonction récursive qui coïncide avec la fonction caractéristique de K. En effet, pour tout x, le γ-programme $q(2x+1,p)$ ne peut être impair, puisqu'il calcule une fonction non-totale, et que tous les γ-programmes impairs calculent des fonctions totales. Par la définition de γ et la sémantique de pr-comp-h, notre affirmation est vérifiée. Nous devons donc conclure que pr-comp-h $\notin \mathsf{UTIL}(\gamma)$, et donc que s-1-1 $\not\models_m$ pr-comp-h.

(d) : Voici maintenant le SP δ.

$$\begin{aligned}\delta(2n) &\stackrel{\mathrm{d}}{=} \phi_n,\\ \delta(2n+1) &\stackrel{\mathrm{d}}{=} \begin{cases}\lambda\langle x,y\rangle.x+1 & \text{si } n \in K,\\ \lambda\langle x,y\rangle.x & \text{autrement.}\end{cases}\end{aligned}$$

Clairement, $\delta \in \mathcal{SPM}$. On vérifie aussi aisément que $\lambda i.2i$ est une instance effective de prog pour δ. Supposons maintenant qu'il existe s, une instance effective de s-1-1 pour δ. Nous affirmons que $\lambda x.\phi_{s(2x+1,0)/2}(0)$ est une fonction récursive qui coïncide avec la fonction caractéristique de K. En effet, pour tout x, le δ-programme $s(2x+1,0)$ ne peut être impair, puisqu'il calcule une fonction constante, et que tous les δ-programmes impairs calculent des fonctions non-constantes. Par la définition de δ et la sémantique de s-1-1, notre

affirmation est vérifiée. Nous devons donc conclure que $\mathsf{s\text{-}1\text{-}1} \notin \mathsf{UTIL}(\delta)$, et donc que $\mathsf{prog} \not\models_m \mathsf{s\text{-}1\text{-}1}$. □

La partie (d) du théorème précédent apparaît comme Théorème 1.3 dans [Ric82] (où il est attribué, sans référence, à Robert Byerly), et contredit une partie de l'énoncé de l'Exercice 2-10 (c) de [Rog67].[1] Nous l'avons obtenue indépendamment. Bien qu'il ne s'agisse pas d'un résultat original, nous estimons que notre preuve présente un intérêt par rapport à celle de [Ric82], car notre SP témoin (δ) est plus simple, et défini de façon plus intuitive.

4.3 po-comp-h contre comp, pr-comp-h, s-1-1 et prog

Nous passons maintenant à la deuxième série de résultats, qui sont en majeure partie des résultats d'indépendance. Nous rappelons au lecteur que les résultats des Sections 4.1, 4.2 et 4.3 sont présentés schématiquement sur la Figure 4.

4.8 Théorème (a) $\mathsf{comp} \models_m \mathsf{po\text{-}comp\text{-}h} \models_m \mathsf{prog}$.

(b) $\{\mathsf{po\text{-}comp\text{-}h}, \mathsf{pr\text{-}comp\text{-}h}\} \not\models_m \mathsf{comp}$.

(c) $\mathsf{prog} \not\models_m \{\mathsf{po\text{-}comp\text{-}h}, \mathsf{s\text{-}1\text{-}1}\}$.

(d) $\mathsf{pr\text{-}comp\text{-}h} \not\models_m \mathsf{po\text{-}comp\text{-}h}$.

(e) $\mathsf{s\text{-}1\text{-}1} \not\models_m \{\mathsf{po\text{-}comp\text{-}h}, \mathsf{pr\text{-}comp\text{-}h}\}$.

(f) $\mathsf{po\text{-}comp\text{-}h} \not\models_m \mathsf{s\text{-}1\text{-}1}$.

(g) $\{\mathsf{po\text{-}comp\text{-}h}, \mathsf{s\text{-}1\text{-}1}\} \not\models_m \mathsf{pr\text{-}comp\text{-}h}$.

Preuve. (a) : Une instance effective de comp est bien sûr SE2. Donc, par la Proposition 4.6 (b), $\mathsf{comp} \models_m \mathsf{po\text{-}comp\text{-}h}$. Riccardi [Ric82] observe qu'un corollaire à la construction de Machtey et Young [MY78, Théorème 3.1.2] est que tout SP maximal possédant une instance SE2 de comp est programmable. Donc, encore par la Proposition 4.6 (b), $\mathsf{po\text{-}comp\text{-}h} \models_m \mathsf{prog}$.

Dans le reste de cette preuve, nous utilisons les SP β, γ et δ de la preuve du Théorème 4.7.

1. L'erreur dans cet exercice est reconnue dans [Rog87], la nouvelle édition de [Rog67].

(d) : Il suffit de montrer que po-comp-h $\notin$ UTIL(β). Supposons qu'il existe q, une instance effective de po-comp-h dans β. Soit f la fonction (a priori partielle) calculée par l'algorithme suivant :

Entrée : n
Algorithme pour f :
1. $z \leftarrow q(pd, 2 \cdot \langle id, n\rangle + 1)$.
2. Retourner 1 si $\phi_{z/2}(0)\downarrow = 2$, et 0 si $\phi_{z/2}(0)\downarrow \neq 2$.

□ **Algorithme**

Un argument essentiellement identique à celui dans la preuve de la partie (b) du Théorème 4.7 montre que f est récursive et coïncide avec la fonction caractéristique de K. Nous concluons donc que po-comp-h $\notin$ UTIL(β).

(e) : Il suffit de montrer que po-comp-h $\notin$ UTIL(γ). Supposons qu'il existe q, une instance effective de po-comp-h dans γ. Soit f la fonction (a priori partielle) calculée par l'algorithme suivant :

Entrée : n
Algorithme pour f :
1. $z \leftarrow q(pd, 2n + 1)$.
2. Retourner 1 si $\phi_{z/2}(0)\downarrow = 2$, et 0 si $\phi_{z/2}(0)\downarrow \neq 2$.

□ **Algorithme**

Nous affirmons que f est récursive et coïncide avec la fonction caractéristique de K. En effet, pour tout n, le z obtenu à l'Étape 1 de l'algorithme est un γ-programme calculant une fonction d'image $\{3\}$ si $n \in K$ et $\{2\}$ autrement. Ce z ne peut être impair, car tous les γ-programmes impairs calculent des fonctions dont l'image est incluse dans $\{0, 1\}$. Donc, par la définition de γ, $\gamma_z = \phi_{z/2}$, et donc, l'Étape 2 se termine toujours et notre affirmation est vérifée. On conclut que po-comp-h $\notin$ UTIL(γ).

(c) : Il suffit de montrer que po-comp-h $\notin$ UTIL(δ). Supposons qu'il existe q, une instance effective de po-comp-h dans δ. Soit f la fonction (a priori partielle) calculée par l'algorithme suivant :

Entrée : n
Algorithme pour f :
1. $z \leftarrow q(pd, 2n + 1)$.
2. Retourner 1 si $\phi_{z/2}(0)\downarrow = 3$, et 0 si $\phi_{z/2}(0)\downarrow \neq 3$.

□ **Algorithme**

Nous affirmons que f est récursive et coïncide avec la fonction caractéristique de K. En effet, pour tout n, le z obtenu à l'Étape 1 de l'algorithme est un δ-programme calculant une fonction d'image $\overline{\{0,1,2\}}$ si $n \in K$ et $\overline{\{0,1\}}$ autrement. Ce z ne peut être impair, car tous les δ-programmes impairs calculent des fonctions dont l'image est soit N, soit $\overline{\{0\}}$. Donc, par la définition de δ, $\delta_z = \phi_{z/2}$, et donc, l'Étape 2 se termine toujours et notre affirmation est vérifiée. On conclut que $\textsf{po-comp-h} \notin \textsf{UTIL}(\delta)$.

(f) : Nous exhibons un SP maximal ε tel que $\textsf{po-comp-h} \in \textsf{UTIL}(\varepsilon)$ et $\textsf{s-1-1} \notin \textsf{UTIL}(\varepsilon)$.

$$\begin{aligned} \varepsilon(2n) &\stackrel{\text{d}}{=} \phi_n \\ \varepsilon(2\cdot\langle a,b\rangle+1) &\stackrel{\text{d}}{=} \begin{cases} \phi_a \circ \lambda\langle x,y\rangle.\left\{\begin{array}{ll} 1 & \text{si } x=0, \\ \uparrow & \text{autrement,} \end{array}\right\} & \text{si } b \in K, \\ \phi_a \circ \lambda\langle x,y\rangle.\left\{\begin{array}{ll} 0 & \text{si } x=0, \\ \uparrow & \text{autrement,} \end{array}\right\} & \text{autrement.} \end{cases} \end{aligned}$$

Clairement, $\varepsilon \in \mathcal{SPM}$. Par une construction très similaire à celle utilisée dans la preuve du Théorème 4.7 pour montrer que $\textsf{pr-comp-h} \in \textsf{UTIL}(\beta)$, on peut facilement montrer que $\textsf{po-comp-h} \in \textsf{UTIL}(\varepsilon)$. Supposons maintenant qu'il existe s, une instance effective de $\textsf{s-1-1}$ pour ε. Nous affirmons qu'alors, $\lambda x.\phi_{s(2\cdot\langle id,x\rangle+1,0)/2}(0)$ est une fonction récursive qui coïncide avec la fonction caractéristique de K. En effet, pour tout x, le ε-programme $z \stackrel{\text{d}}{=} s(2\cdot\langle id,x\rangle+1,0)$ ne peut être impair, car il calcule une fonction totale, et tous les ε-programmes impairs calculent des fonctions non-totales. De plus, ε_z est la fonction constante 1 si $x \in K$, et 0 autrement. Par définition de ε, $\varepsilon_z = \phi_{z/2}$, et notre affirmation est vérifiée.

(g) : Nous exhibons ζ, un SP maximal tel que $\{\textsf{po-comp-h}, \textsf{s-1-1}\} \subseteq \textsf{UTIL}(\zeta)$ et $\textsf{pr-comp-h} \notin \textsf{UTIL}(\zeta)$.

$$\begin{aligned} \zeta(2n) &\stackrel{\text{d}}{=} \phi_n, \\ \zeta(2\cdot\langle a,b\rangle+1) &\stackrel{\text{d}}{=} \begin{cases} \phi_a \circ \lambda z.1 & \text{si } b \in K, \\ \phi_a \circ \lambda z.0 & \text{autrement.} \end{cases} \end{aligned}$$

Clairement, $\zeta \in \mathcal{SPM}$. Par une construction très similaire à celle utilisée dans la preuve du Théorème 4.7 pour montrer que $\textsf{pr-comp-h} \in \textsf{UTIL}(\beta)$, on peut facilement montrer que $\textsf{po-comp-h} \in \textsf{UTIL}(\zeta)$. On vérifie par ailleurs

aisément que la fonction s définie par

$$s(i,y) \stackrel{\mathrm{d}}{=} \begin{cases} 2 \cdot s_\phi(i/2, y) & \text{si } i \text{ est pair,} \\ i & \text{autrement} \end{cases}$$

est une instance effective de $\mathsf{s\text{-}1\text{-}1}$ pour ζ. Supposons maintenant qu'il existe q, une instance effective de $\mathsf{pr\text{-}comp\text{-}h}$ pour ζ. Soit p un ϕ-programme pour $\lambda z.[0$ si $z = 0$; $\uparrow$ autrement$]$. Nous affirmons qu'alors, $\lambda x.\phi_{q(2\cdot\langle id,x\rangle+1,p)/2}(0)$ est une fonction récursive qui coïncide avec la fonction caractéristique de K. En effet, pour tout x, le ζ-programme $z \stackrel{\mathrm{d}}{=} q(2 \cdot \langle id, x\rangle + 1, p)$ ne peut être impair, car il calcule une fonction de domaine $\{0\}$, et tous les ζ-programmes impairs calculent des fonctions de domaine N ou $\emptyset$. De plus, $\zeta_z(0)$ vaut 1 si $x \in K$, et 0 autrement. Par définition de ζ, $\zeta_z = \phi_{z/2}$, et notre affirmation est vérifiée.

(b) : Nous exhibons η, un SP maximal tel que $\{\mathsf{pr\text{-}comp\text{-}h}, \mathsf{po\text{-}comp\text{-}h}\} \subseteq \mathsf{UTIL}(\eta)$ et $\mathsf{comp} \notin \mathsf{UTIL}(\eta)$. Nous utilisons une définition auxiliaire. Pour tout j, on définit

$$f_j \stackrel{\mathrm{d}}{=} \begin{cases} \lambda z. \left\{ \begin{array}{ll} \uparrow & \text{si } z = j, \\ z & \text{autrement,} \end{array} \right\} & \text{si } j \in K, \\ \lambda z.z & \text{autrement.} \end{cases}$$

On remarque que pour tout j, f_j est soit la restriction de l'identité à $\overline{\{j\}}$, soit l'identité, suivant respectivement que $j \in K$ ou non. Pour tout j, donc, f_j est partielle récursive.

On définit alors η comme suit :

$$\begin{array}{rcl} \eta(2n) & \stackrel{\mathrm{d}}{=} & \phi_n, \\ \eta(2 \cdot \langle i, j, k\rangle + 1) & \stackrel{\mathrm{d}}{=} & \phi_i \circ f_j \circ \phi_k. \end{array}$$

Clairement, $\eta \in \mathcal{SPM}$. Par des constructions très similaires à celle utilisée dans la preuve du Théorème 4.7 pour montrer que $\mathsf{pr\text{-}comp\text{-}h} \in \mathsf{UTIL}(\beta)$, on peut facilement montrer que $\{\mathsf{pr\text{-}comp\text{-}h}, \mathsf{po\text{-}comp\text{-}h}\} \subseteq \mathsf{UTIL}(\eta)$. Supposons maintenant qu'il existe c, une instance effective de comp pour ψ. Définissons la fonction g comme suit.

$$\begin{array}{lcl} g(x,0) & \stackrel{\mathrm{d}}{=} & 2 \cdot \langle id, x, id\rangle + 1, \\ g(x, n+1) & \stackrel{\mathrm{d}}{=} & c(2 \cdot \langle id, n, id\rangle + 1, g(x,n)), \quad \text{pour tout } n. \end{array}$$

Il est clair que g est récursive. Par commodité, si x est impair et si i, j et k sont tels que $x = 2 \cdot \langle i, j, k \rangle + 1$, alors on dénotera i, j et k par, respectivement, $x[1]$, $x[2]$ et $x[3]$.

Nous avons le lemme suivant.

4.9 Lemme *Pour tout x, tout n et tout m,*

$$m \notin \mathbf{dom}(\eta_{g(x,n)}) \iff m \in K \cap (\{0, 1, \dots, n-1\} \cup \{x\}).$$

Preuve. Il suffit de remarquer que

$$\eta_{g(x,n)} = f_{n-1} \circ f_{n-2} \circ \cdots \circ f_0 \circ f_x.$$

Par les remarques faites plus haut sur les f_j, le lemme est établi.

□ **Lemme 4.9**

4.10 Corollaire (du Lemme 4.9) *Pour tout x, pour tout n et pour tout $m \in \{0, 1, \dots, n-1\} \cup \{x\}$, soit $z = g(x, n)$. Alors, si z est pair,*

$$m \notin K \iff \phi_{z/2}(m) \text{ est défini}$$

et, si z est impair,

$$m \notin K \iff \phi_{z[1]} \circ f_{z[2]} \circ \phi_{z[3]}(m) \text{ est défini.}$$

Preuve. Immédiat par le lemme et la définition de η. □ **Corollaire 4.10**

Revenons à la preuve du théorème. Nous traitons séparément les cas où il existe un $m \notin K$ tel que

$$\forall n,\ g(m, n) \text{ est impair et } \phi_{g(m,n)[3]}(m) = g(m, n)[2]$$

et où il n'en existe pas. Dans les deux cas, nous exhibons une procédure effective pour énumérer $\overline{K}$, ce qui constitue une contradiction du fait que $\overline{K}$ n'est pas r.é.

Premier cas Soit $m \notin K$ tel que

$$(\forall n)[g(m,n) \text{ est impair et } \phi_{g(m,n)[3]}(m) = g(m,n)[2]].$$

On remarque la chose suivante.

4.11 Lemme *Pour tout n, $f_{g(m,n)[2]}$ est l'identité.*

Preuve. Puisque $m \notin K$, par le Corollaire 4.10, on a que pour tout n, en posant $z = g(m,n)$,

$$\phi_{z[1]} \circ f_{z[2]} \circ \phi_{z[3]}(m) \text{ est défini.}$$

Or, par hypothèse sur m, $\phi_{z[3]}(m) = z[2]$. Donc, $f_{z[2]}(z[2])$ est défini, d'où nous concluons, par la définition des f_j, que $f_{z[2]}$ est l'identité.

$\square$ **Lemme 4.11**

Revenons à la preuve du théorème. Nous affirmons que, pour énumérer $\overline{K}$, il suffit de faire, en parallèle pour chaque $x \in N$ (par queue de colombe), la suite d'étapes suivante.

1. $z \leftarrow g(m, x+1)$.
2. $i \leftarrow z[1]$; $j \leftarrow z[2]$; $k \leftarrow z[3]$.
3. Si $\phi_i(\phi_k(x)\downarrow)\downarrow$, lister x.

Montrons que cette procédure énumère bien tous les membres de $\overline{K}$ et aucun membre de K.

L'Étape 1 se termine toujours, du fait que g est totale. Par hypothèse sur m, $g(m, x+1)$ est nécessairement impair, et l'Étape 2 est justifiée. L'Étape 3 est donc toujours exécutée et, par le Corollaire 4.10,

$$x \notin K \Longleftrightarrow \phi_i \circ f_j \circ \phi_k(x) \text{ est défini.}$$

Or, par le Lemme 4.11, f_j est l'identité. Donc, l'Étape 3 se termine ssi $x \notin K$, et x sera listé ssi $x \notin K$.

Deuxième cas Supposons qu'il n'existe pas de $m \notin K$ tel que

$$(\forall n)[g(m,n) \text{ est impair et } \phi_{g(m,n)[3]}(m) = g(m,n)[2]].$$

En d'autres termes, posons l'hypothèse que

$$(\forall m \notin K)(\exists n)[g(m,n) \text{ est pair } \vee \phi_{g(m,n)[3]}(m) \neq g(m,n)[2]].$$

Nous affirmons qu'alors, on peut énumérer $\overline{K}$ en effectuant, en parallèle pour chaque $x \in N$ (par queue de colombe), la suite d'étapes suivante.

1. $n \leftarrow 0$; aller à l'Étape 3.
2. $n \leftarrow n+1$.
3. $z \leftarrow g(x,n)$. Si z est pair, aller à l'Étape 4. Sinon, aller à l'Étape 5.
4. (z est pair) : Si $\phi_{z/2}(x)\downarrow$, lister x puis arrêter.
5. (z est impair) : $r \leftarrow \phi_{z[3]}(x)$. Si $r = z[2]$, retourner à l'Étape 2 ; sinon, passer à l'étape suivante.
6. (z impair et $r = \phi_{z[3]}(x) \neq z[2]$) : Si $\phi_{z[1]}(r)\downarrow$, lister x.

Montrons que cette procédure énumère bien tous les membres de $\overline{K}$ et aucun membre de K.

L'Étape 4 n'est exécutée que si un z pair est obtenu lors d'un passage à l'Étape 3. Alors, par le Corollaire 4.10,

$$x \notin K \iff \phi_{z/2}(x) \text{ est défini.}$$

Ainsi, nous listerons x à l'Étape 4 ssi $x \notin K$.

L'Étape 6 n'est exécutée que si un z impair est obtenu lors d'un passage à l'Étape 3, et si $r = \phi_{z[3]}(x) \neq z[2]$. On note qu'alors, par la définition des f_j, $f_{z[2]}(r) = r$. Donc,

$$\phi_{z[1]} \circ f_{z[2]} \circ \phi_{z[3]}(x) = \phi_{z[1]} \circ f_{z[2]}(r) = \phi_{z[1]}(r),$$

et donc,

$$\phi_{z[1]} \circ f_{z[2]} \circ \phi_{z[3]}(x) \text{ est défini } \iff \phi_{z[1]}(r) \text{ est défini,}$$

d'où, par le Corollaire 4.10,

$$x \notin K \iff \phi_{z[1]}(r) \text{ est défini.}$$

Ainsi, nous listerons x à l'Étape 6 ssi $x \notin K$.

Comme x ne peut être listé ailleurs qu'aux Étapes 4 et 6, nous concluons que la procédure n'énumère aucun membre de K. D'autre part, par hypothèse,

l'Étape 6 ou l'Étape 4 est éventuellement atteinte pour tout $x \notin K$, et nous concluons que la procédure énumère tous les membres de $\overline{K}$. Ceci contredit le fait que $\overline{K}$ n'est pas r.é., et nous permet de conclure que $\mathsf{comp} \notin \mathsf{UTIL}(\eta)$.

$\square$ **Théorème 4.11**

Notons que Riccardi a montré (essentiellement) que $\mathsf{s\text{-}1\text{-}1} \not\models_m \mathsf{po\text{-}comp\text{-}h}$ [Ric82, Théorème 3.2]. La partie (f) du théorème précédent répond à une question ouverte de [Ric82] (p. 294).

4.4 prog contre les RS de Myhill-Shepherdson

Toute RS qui garantit prog dans les SP maximaux garantit forcément acc dans les SP *exécutables* maximaux. Ainsi, comp, pr-comp-h, po-comp-h, s-1-1 de même que la possession d'une instance SE1 *ou* SE2 de s-1-1 garantissent l'acceptabilité dans les SP *exécutables* maximaux. Il est immédiat de montrer que pour n'importe quel SP acceptable ϕ', la RS $(\mathcal{R}, \mathit{VRAI})$ où pour tout ψ, p et q, $\mathcal{R}(\psi, p, q) \Leftrightarrow [\psi_q = \phi'_p]$, qui clairement garantit acc dans les SP exécutables, est *équivalente* à prog dans les SP maximaux.

Toute RS garantissant acc dans les SP exécutables maximaux *doit*-elle donc garantir prog dans les SP maximaux ? Royer a défini (comme structure de contrôle) la RS $\mathsf{strange} \stackrel{\mathrm{d}}{=} (\mathcal{R}, \mathit{VRAI})$, où pour tout ψ, p et q, $\mathcal{R}(\psi, p, q) \Leftrightarrow [\psi_q = \phi(\psi_p(0))]$ [Roy87, Définition 4.2.1]. On vérifie facilement que strange est de Myhill-Shepherdson (Définition 2.54) et donc, en particulier, purement extensionnelle sans récursion au sens de Royer. Royer a montré que $\mathsf{strange} \models_{em} \mathsf{acc}$ [Roy87, Théorème 4.2.2 (b)]. Le théorème suivant implique que $\mathsf{strange} \not\models_m \mathsf{prog}$, et la question ci-dessus reçoit donc une réponse négative. La preuve offre des similarités avec la preuve de [Roy87, Théorème 4.2.2 (c)], bien qu'elle généralise clairement celle-ci.

4.12 Théorème *Aucune RS de Myhill-Shepherdson ne garantit à elle seule* prog *dans les SP maximaux.*

Preuve. Soit rs une RS de Myhill-Shepherdson. Nous construisons ψ, un SP *non-programmable* maximal et ayant une instance effective de rs.

Comme rs est de Myhill-Shepherdson, il existe un opérateur récursif Θ tel que pour toute f et tout ψ, f est une instance de rs dans ψ ssi pour tout

p, $\psi_{f(p)} = \Theta(\psi_p)$. Soit η un SP à puissance de calcul maximale tel que $A \stackrel{\mathrm{d}}{=} \{p \mid \eta_p \neq \lambda z.\uparrow\}$ est immunisé. Un tel SP existe par des arguments classiques (non-constructifs) en théorie de la récursion. On définit alors ψ comme suit :

$$\begin{aligned} \psi(2n) &\stackrel{\mathrm{d}}{=} \eta_n \\ \psi(2n+1) &\stackrel{\mathrm{d}}{=} \Theta(\psi_n) \end{aligned}$$

Clairement, $\lambda p.[2p+1]$ est une instance effective de rs dans ψ. Supposons maintenant que ψ soit programmable, et soit t une fonction de programmation pour ψ. Nous traitons séparément les cas où $\Theta(\emptyset) \neq \emptyset$ ($= \lambda z.\uparrow$) et où $\Theta(\emptyset) = \emptyset$.

Supposons que $\Theta(\emptyset) \neq \emptyset$. Alors, il existe $(x, y) \in \Theta(\emptyset)$. Par monotonicité des opérateurs récursifs, $(x, y) \in \Theta(\alpha)$ pour toute $\alpha \in \mathcal{P}art\mathcal{U}n$ et en particulier, pour toute $\alpha \in \mathcal{P}art\mathcal{R}ec\mathcal{U}n$. Alors, clairement, $(x, y) \in \psi_{2n+1}$ pour tout n. Soit g une fonction récursive telle que pour tout m, $\phi_{g(m)} = \{(x+1, m)\}$. Clairement, tous les ψ-programmes $t(g(m))$, $m \in N$ doivent être distincts, et doivent être de la forme $2n$ pour un certain $n \in A$. L'ensemble $\{t(g(m))/2 \mid m \in N\}$ est donc un sous-ensemble infini récursivement énumérable de A, une contradiction puisque A est immunisé.

Supposons maintenant que $\Theta(\emptyset) = \emptyset$. Définissons les fonctions récursives q et c comme suit :

$$\begin{aligned} q &\stackrel{\mathrm{d}}{=} \lambda p.[\text{l'exposant de 2 dans la factorisation de } p+1], \\ c &\stackrel{\mathrm{d}}{=} \lambda p.[(p+1)/2^{q(p)} - 1]. \end{aligned}$$

Informellement, pour tout p, $q(p)$ donne le nombre de 1 consécutifs à partir de la position la moins significative dans la représentation binaire de p, et $c(p)$ donne la valeur de ce qui reste si on élimine ces bits. Pour tout p, on appelle $c(p)$ le *corps* de p, et $q(p)$ la *queue* de p. On note que le corps d'un entier quelconque est toujours pair et qu'un corps et une queue spécifiques déterminent uniquement un entier ; on note également que pour tout ψ-programme p, $\psi_p = \Theta^{q(p)}(\psi_{c(p)}) = \Theta^{q(p)}(\eta_{c(p)/2})$. Puisque $\Theta(\emptyset) = \emptyset$, on en déduit que le corps de tout ψ-programme calculant autre chose que la fonction indéfinie partout est nécessairement de la forme $2n$ pour un certain $n \in A$.

Soit maintenant g une fonction récursive telle que $\mathbf{image}(g) = \{p \mid \phi_p \neq \emptyset\}$, un ensemble récursivement énumérable infini. Considérons l'ensemble $B \stackrel{\mathrm{d}}{=}$

$\{t(g(m)) \mid m \in N\}$, qui est manifestement aussi un ensemble récursivement énumérable. Remarquons qu'aucun ψ-programme dans B ne calcule la fonction indéfinie partout et qu'alors, $C \stackrel{\mathrm{d}}{=} \{c(x) \mid x \in B\}$ est nécessairement fini (quoique bien sûr, non-vide) : en effet, si C était infini, $\{p/2 \mid p \in C\}$ serait un sous-ensemble récursivement énumérable infini de A, ce qui contredirait l'hypothèse d'immunité de A. Soit $C = \{c_1 < \cdots < c_k\}$.

Imaginons maintenant un tableau de k colonnes par une infinité dénombrable de lignes. Les colonnes sont indexées par $\underline{k}$ et les lignes par N. La position i, j contient le ψ-programme de corps c_i et de queue j. Il est clair que B est entièrement contenu (peut-être proprement) dans notre tableau. Nous montrons que l'ensemble des ψ-programmes de notre tableau qui calculent $\lambda z.0$, appelons-le D, est récursif. Notons que pour tout $p \in B$, si $\psi_p = \lambda z.0$, alors $p \in D$.

Soit $D_i,\ i \in \underline{k}$ l'intersection de D et de la i-ième colonne de notre tableau. Si nous montrons que pour chaque $i \in \underline{k}$, D_i est récursif, alors clairement, il s'ensuivra que D est récursif. Soit donc $i \in \underline{k}$. Si D_i est vide ou un singleton, alors il est clairement récursif. Si D_i contient plus d'un élément, alors soient a et b ($a < b$) les deux plus petits éléments. Notons que $\psi_b = \Theta^{q(b)-q(a)}(\psi_a)$ et rappelons-nous que $\psi_b = \psi_a = \lambda z.0$. Donc, $\Theta^{q(b)-q(a)}(\lambda z.0) = \lambda z.0$, et pour tout j tel que $1 \leq j < q(b) - q(a)$, $\Theta^j(\lambda z.0) \neq \lambda z.0$. Il s'ensuit que D_i contient exactement les ψ-programmes de corps c_i et de queues $q(a) + n \cdot (q(b) - q(a)),\ n \in N$; ceci en fait clairement un ensemble récursif. Nous concluons que D est récursif.

Considérons la procédure d'énumération suivante : pour tout n, lister $g(n)$ ssi $t(g(n)) \in D$. On vérifie facilement que cette procédure énumère $\{p \mid \phi_p = \lambda z.0\}$, lequel, par le Théorème de Rice [HU79], n'est pas r.é. : une contradiction. Nous devons donc conclure que ψ n'est pas programmable.

$\square$ **Théorème 4.12**

Le Théorème 4.12 est d'une certaine façon le plus fort possible. En effet, comp, une RS purement extensionnelle sans récursion au sens de Royer, mais à *deux* arguments, garantit prog dans les SP maximaux, et prog elle-même est extensionnelle sans récursion au sens de Royer, à un argument, mais non *purement* extensionnelle. Aussi, on connaît plusieurs ensembles de *deux* RS de Myhill-Shepherdson qui garantissent prog dans les SP maximaux (e.g., [Roy87, Théorème 4.1.4]).

Le théorème suivant montre qu'en fait, $\mathsf{strange}$ est indépendante de prog.

4.13 Théorème $\mathsf{prog} \not\models_m \mathsf{strange}$.

Preuve. Soient p_0 et p_1 des ϕ-programmes pour respectivement $\lambda z.[0 \text{ si } x = 0\,;\ \uparrow \text{ autrement}]$ et $\lambda z.[1 \text{ si } x = 0\,;\ \uparrow \text{ autrement}]$. Définissons maintenant ψ comme suit :

$$\begin{aligned} \psi(2n) &\stackrel{\mathrm{d}}{=} \phi_n, \\ \psi(2n+1) &\stackrel{\mathrm{d}}{=} \begin{cases} \lambda z.p_1 & \text{si } n \in K, \\ \lambda z.p_0 & \text{autrement.} \end{cases} \end{aligned}$$

Clairement, ψ est programmable via $\lambda n.2n$, et est donc maximal. Supposons maintenant qu'il existe s, une instance effective de $\mathsf{strange}$ dans ψ. Nous affirmons que $\phi_{s(2n+1)/2}(0)$ est récursive et coïncide avec la fonction caractéristique de K. En effet, par définition de $\mathsf{strange}$, pour tout n, $\psi_{s(2n+1)} = \phi(\psi_{2n+1}(0))$. Or, par définition de ψ, $\psi_{2n+1}(0)$ est p_1 si $n \in K$ et p_0 autrement. On a donc $\psi_{s(2n+1)} = \phi(p_1)$ si $n \in K$ et $\psi_{s(2n+1)} = \phi(p_0)$ autrement. Dans les deux cas, $s(2n+1)$ ne peut être un ψ-programme impair, car tous les ψ-programmes impairs calculent des fonctions totales, et $\phi(p_0)$ et $\phi(p_1)$ sont toutes deux non-totales. Comme $s(2n+1)$ est pair, alors, par définition de ψ, $\psi_{s(2n+1)} = \phi_{s(2n+1)/2}$. D'autre part, si $n \in K$, alors $\psi_{s(2n+1)}(0) = \phi_{p_1}(0) = 1$, et autrement, $\psi_{s(2n+1)}(0) = \phi_{p_0}(0) = 0$. Notre affirmation est donc vérifiée. Comme elle contredit la non-récursivité de K, on doit conclure que $\mathsf{strange} \notin \mathsf{UTIL}(\psi)$, et donc que $\mathsf{prog} \not\models_m \mathsf{strange}$. □

4.5 Discussion

Le fait mentionné plus haut (dû à Royer) que $\mathsf{strange} \models_{em} \mathsf{acc}$, combiné à notre Théorème 4.12, montre que toute RS garantissant acc dans $\mathcal{SPEM}$ ne garantit pas nécessairement prog dans $\mathcal{SPM}$. Notons cependant que, par notre Théorème 4.13, $\mathsf{strange}$ n'est *pas* strictement plus faible que prog, simplement indépendante d'elle. Nous soupçonnons que parmi les RS d'une classe assez vaste, mettons les RVE à prédicat *VRAI*, aucune ne peut à la fois garantir acc dans $\mathcal{SPEM}$ et être *strictement plus faible que* prog dans $\mathcal{SPM}$. Autrement dit, nous croyons que pour toute RVE rs à prédicat *VRAI*, si $\mathsf{prog} \models_m \mathsf{rs}$ et $\mathsf{rs} \models_{em} \mathsf{acc}$, alors $\mathsf{rs} \models_m \mathsf{prog}$.

On connaît certaines choses sur les versions SE1 et SE2 de s-1-1. Riccardi a montré [Ric82, Théorème 1.2] que, pour les SP maximaux, la possession d'une instance SE2 de s-1-1 est équivalente à la possession d'une instance effective de prog. D'autre part, la preuve du Théorème 4.1.4 dans [Roy87] donne comme corollaire que la possession d'une instance SE1 de s-1-1 garantit la possession d'une instance effective de prog dans les SP maximaux. Dans notre preuve de la partie (d) du Théorème 4.7, la construction effectuée pour démontrer que s-1-1 $\notin$ UTIL(δ) n'utilise la prétendue instance effective de s-1-1 pour δ *qu'avec un deuxième argument fixe*. On peut donc déduire que δ ne possède pas d'instance SE1 de s-1-1, et donc que la programmabilité ne garantit pas la possession d'une instance SE1 de s-1-1, pour les SP maximaux. (Nous savons aussi exhiber un SP maximal possédant une instance SE1 de s-1-1, mais pas d'instance effective de s-1-1 ; la possession d'une instance effective de s-1-1 est donc une propriété strictement plus "forte" que la possession d'une instance SE1 de s-1-1.)

On pourrait espérer, a priori, "descendre" les résultats d'indépendance de ce chapitre au contexte des SP exécutables maximaux, en les transformant en des résultats d'indépendance de complexité des instances. Considérons par exemple le SP δ de la preuve du Théorème 4.7 (d). La fonction $\lambda z.2z$, calculable en temps linéaire, est une instance de prog pour δ. Par contre, δ ne possède aucune instance effective de s-1-1. Supposons maintenant qu'on remplace l'ensemble K intervenant dans la définition de δ par un ensemble qui, au lieu d'être non-récursif, est récursif mais très difficile (e.g., pour lequel toute procédure de décision exige un temps de calcul double-exponentiel). Appelons le SP résultant δ'. Notez que δ' est exécutable (et donc, acceptable), et possède nécessairement une instance effective de s-1-1. Par contre, comme K est difficile, on pourrait s'attendre à ce que toutes les instances effectives de s-1-1 dans δ' exigent beaucoup de temps de calcul. Cependant, tel n'est pas le cas. On montre facilement qu'une instance dans $\mathcal{L}$*intime* de prog dans un SP exécutable maximal garantit une instance dans $\mathcal{L}$*intime* de s-1-1 (et donc aussi, par le Théorème 3.4, de n'importe quelle RS dans LC-RVE). Si on compare l'argument montrant que δ ne possède aucune instance effective de s-1-1 avec celui montrant que δ' possède une instance dans $\mathcal{L}$*intime* de s-1-1, on réalise qu'en essence, la raison pour laquelle δ' possède une instance dans $\mathcal{L}$*intime* de s-1-1 est que, dans un SP acceptable, la complexité d'un calcul dont la raison d'être est de trouver un programme peut toujours être "repoussée" dans la complexité du programme trouvé. Grosso modo et informellement, au lieu d'effectuer un calcul coûteux et de retourner un programme A, on retourne un programme B qui d'abord effectue le

calcul coûteux, trouvant par le fait même le programme A, puis ensuite, fait exécuter ce programme (A) par un interprète du SP. Ainsi, bien que pour δ, la résolution du problème d'appartenance à K *doive* être effectuée par une prétendue instance effective de s-1-1, pour δ', elle peut être reportée au moment de l'exécution du programme qui est retourné comme sortie de l'instance de s-1-1.

Chapitre 5

Problèmes de décision et de recherche sur les systèmes de programmation

Les preuves d'interrelations de garantie entre deux RS sont en général non-constructives, en ce sens qu'elles ne permettent pas de conclure que la connaissance d'une instance effective de la RS "garantissante" est suffisante pour "construire" une instance effective de la RS "garantie", même si elles établissent que l'*existence* d'une instance effective de la première implique l'*existence* d'une instance effective de la seconde. Dans ce chapitre, nous démontrons la non-constructivité de l'interrelation spécifique $\mathsf{s\text{-}1\text{-}1} \models_{em} \mathsf{prog}$. Nous identifions aussi certaines interrelations constructives. Nous étudions également l'indécidabilité de certains problèmes de décision sur les SP et la difficulté de trouver une fonction de programmation (instance effective de prog) pour un SP acceptable à partir d'un *interprète* de ce SP. Ce dernier problème est intuitivement très lié au problème de constructivité d'une instance d'une RS à partir d'une instance d'une autre RS dans le même SP, et peut être vu comme représentant un problème "d'apprentissage de la programmation".

Pour parler de problème de décision ou de recherche sur les SP, nous prenons comme "exemplaires" les interprètes des SP, i.e., les ϕ-programmes qui calculent leur fonction universelle (unaire). Nous ne parlerons donc que de problèmes sur les SP *exécutables*, puisque ce sont les seuls qui sont représen-

tés par un "exemplaire". Un autre corollaire à notre approche est que nous n'avons pas un représentant unique pour chaque SP, puisqu'il existe une infinité d'interprètes distincts pour chaque SP exécutable.

Il serait possible d'introduire une notion d'*interprète avec oracle*, et de pouvoir ainsi avoir des exemplaires pour *certains* SP non-exécutables. Nous n'avons cependant pas suivi cette approche : notre but étant d'étudier des questions de constructivité, il nous apparaît naturel de nous limiter aux exemplaires "réalistes", et seuls les interprètes sans oracle peuvent prétendre à ce titre.

Tout ϕ-programme p est un interprète pour le SP ψ défini par $\psi(q) \stackrel{\mathrm{d}}{=} \lambda z.\phi_p(\langle q, z\rangle)$. Ce SP n'a bien sûr pas nécessairement puissance de calcul maximale. Nous introduisons trois classes intéressantes d'interprètes.

5.1 Définition (a) $\mathit{IM} \stackrel{\mathrm{d}}{=} \{i \mid \phi_i$ est la fonction universelle unaire d'un SP maximal$\}$.

(b) $\mathit{IA} \stackrel{\mathrm{d}}{=} \{i \mid \phi_i$ est la fonction universelle unaire d'un SP acceptable$\}$.

(c) $\mathit{IMNA} \stackrel{\mathrm{d}}{=} \mathit{IM} - \mathit{IA} = \{i \mid \phi_i$ est la fonction universelle unaire d'un SP maximal non-acceptable$\}$.

Pour commodité, nous appellerons les membres de *IM* les interprètes "maximaux", ceux de *IA* les interprètes "acceptables" et ceux de *IMNA* les interprètes "maximaux non-acceptables".

5.2 Notation Pour tout e, ϕ^e dénote le SP (exécutable) dont la fonction universelle unaire est ϕ_e, i.e., le SP ψ tel que $\widehat{\psi} = \phi_e$.

Nous parlerons souvent de l'action "d'inférer" une instance effective d'une RS dans un SP à partir de certaines autres informations sur le SP (e.g., un interprète). Par "inférer" une instance effective d'une RS, nous voulons dire "trouver un ϕ-programme" pour une telle fonction. Si la RS en question est à 1 entrée, alors ses instances effectives sont unaires, et sont donc calculables en ϕ directement. Autrement, cependant, les instances effectives de la RS sont d'arité supérieure à 1, et ne sont pas directement calculables en ϕ. Pour contourner ce petit problème, par convention, "trouver un ϕ-programme" pour une instance effective d'une RS à plus d'une entrée signifiera "trouver un ϕ-programme" pour la version unarisée d'une telle fonction (§ 1.3.5).

La plupart des résultats de ce chapitre sont des résultats négatifs. Parfois, nous montrons l'infaisabilité de certaines choses même avec un oracle dans le problème d'arrêt (i.e., l'ensemble K). Rappelons que ϕ_i^K dénote la fonction, partielle récursive dans le problème d'arrêt, calculée par la machine de Turing, avec oracle dans K, dont le code est i ; et que $W_i^K \stackrel{\mathrm{d}}{=} \mathbf{dom}(\phi_i^K)$ (voir § 1.3.7). Nous utilisons les faits suivants [RoyCas] :

5.3 Fait Il existe g, une fonction récursive, telle que pour tout i, $\lambda z.[\lim_{k\to\infty} g(i,z,k)] = \phi_i^K$, où la notion de limite est définie de la façon usuelle et où on dit que $\lambda z.[\lim_{k\to\infty} g(i,z,k)]$ diverge sur x ssi $\lim_{k\to\infty} g(i,x,k)$ n'existe pas. On note que, pour tout i, $W_i^K = \{z \mid \lim_{k\to\infty} g(i,z,k) \text{ existe}\}$.

5.4 Fait Pour toute fonction partielle récursive dans le problème d'arrêt α, il existe une fonction récursive g telle que $\lambda x.[\lim_{y\to\infty} g(x,y)] = \alpha$.

5.1 Quelques problèmes de décision

Supposons qu'on nous soumet un ordinateur quelconque. Pouvons-nous déterminer si cet ordinateur peut calculer toutes les fonctions partielles récursives ? Déterminer s'il existe un compilateur PASCAL pour cet ordinateur ? Si on nous *garantit* que l'ordinateur soumis peut calculer toutes les fonctions partielles récursives, alors pouvons-nous déterminer s'il existe un compilateur PASCAL pour lui ? Nous attaquons ces questions dans le cadre formel des SP.

Par le Théorème de Rice [HU79], aucun de *IA*, *IM* et *IMNA* n'est même "semi-décidable" (r.é. non-récursif). Mais peut-on en dire plus ? Nous montrons ici que *IA*, *IM* et *IMNA* sont tous trois Σ_3^0-ardus (selon la réducibilité multivoque récursive, dénotée $\leq_m$ [Rog87]). Comme bornes supérieures d'indécidabilité, nous montrons que $IA \in \Sigma_3^0$ (donc, *IA* est Σ_3^0-complet) et que *IM* et *IMNA* sont dans Π_4^0. Nous ne savons pas si *IM* ou *IMNA* sont Π_4^0-complets. Si *IM* était Π_4^0-complet, il s'agirait d'un des exemples les plus naturels d'un tel ensemble.

Nous utiliserons la relation T définie comme suit :

$$T(p,x,y,t) \stackrel{\mathrm{d}}{\Longleftrightarrow} \phi_p(x){\downarrow} = y \text{ en au plus } t \text{ étapes.}$$

Il s'agit clairement d'une relation récursive [Rog87].

La définition "naturelle" de IM le place dans Π^0_4. En effet,

$$IM = \{a \mid (\forall p)(\exists q)(\forall x,y,t)(\exists t')[(t' \geq t) \ \& \\ [T(p,x,y,t') \Leftrightarrow T(a,\langle q,x\rangle,y,t')]]\}.$$

Placer $IMNA$ dans Π^0_4 demande un peu plus de travail.

5.5 Théorème $IMNA \in \Pi^0_4$.

Preuve. Nous donnons un argument informel montrant que $IMNA$ est l'ensemble des a satisfaisant

$$\begin{aligned}&(\forall p,r)(\exists q,i,j,x,y)(\forall d,v,w,t)(\exists t')\\ &\quad [(t' \geq t) \ \& \ [T(p,v,w,t') \Leftrightarrow T(a,\langle q,v\rangle,w,t')] \ \& \\ &\quad\quad [\neg T(r,i,d,t) \vee \\ &\quad\quad\quad [T(r,i,j,t') \ \& \ [T(i,x,y,t') \not\Longleftrightarrow T(a,\langle j,x\rangle,y,t')]]]].\end{aligned}$$

La compréhension de cette formule est facilitée si l'on attache aux variables une signification intuitive, comme suit : p est un ϕ-programme quelconque ; q est un ϕ^a-programme qui doit lui être équivalent, i.e., concorder sur toutes les paires argument–résultat (v,w). Pour tout p, il doit y avoir un tel q : ceci assure que a est un interprète maximal.

La variable accessoire d sert à exprimer le fait qu'un calcul diverge.

La variable r dénote un ϕ-programme qui "essaie" de traduire des ϕ-programmes en ϕ^a-programmes. Tout r doit échouer dans cette tâche, c'est-à-dire qu'il doit exister un certain i sur lequel ϕ_r, soit diverge (dans quel cas, $(\forall d,t)[\neg T(r,i,d,t)]$), soit retourne un j (c'est-à-dire, $(\forall t)(\exists t')[(t' \geq t) \ \& \ T(r,i,j,t')]$) tel qu'il existe un x sur lequel ϕ_i et ϕ^a_j sont en désaccord. Maintenant, on vérifie que ϕ_i et ϕ^a_j sont en désaccord sur x ssi $(\exists y)(\forall t)(\exists t')[(t' \geq t) \ \& \ [T(i,x,y,t') \not\Longleftrightarrow T(a,\langle j,x\rangle,y,t')]]$. □

5.6 Théorème $IA \in \Sigma^0_3$.

Preuve. Pour tout i, $i \in IA$ ssi il existe une fonction récursive t telle que pour tout p, $\lambda x.[\phi_i(\langle t(p),x\rangle)] = \phi_p$. On voit facilement que ceci est équivalent à

$$(\exists c)(\forall p)(\exists q)(\forall x,y)[\phi_c(p)\downarrow = q \ \& \ [\phi_p(x)\downarrow = y \Longleftrightarrow \phi_i(\langle q,x\rangle)\downarrow = y]]. \quad (5.1)$$

On peut récrire la sous-expression

$$(\exists q)(\forall x,y)[\phi_c(p)\!\downarrow = q \ \&\ [\phi_p(x)\!\downarrow = y \iff \phi_i(\langle q,x\rangle)\!\downarrow = y]]$$

de (5.1) comme

$$\begin{aligned}(\exists q)(\forall x,y,t)(\exists t')[\phi_c(p)\!\downarrow = q \ \&\ (t' \geq t)\ \&\\ [T(p,x,y,t') \iff T(i,\langle q,x\rangle,y,t')]],\end{aligned} \tag{5.2}$$

qui est équivalent à

$$\begin{aligned}(\forall x,y,t)(\exists q)(\exists t')[(t' \geq t)\ \&\ \phi_c(p)\!\downarrow = q\ \&\\ [T(p,x,y,t') \iff T(i,\langle q,x\rangle,y,t')]]\end{aligned} \tag{5.3}$$

((5.2) implique (5.3) par simple calcul des prédicats ; (5.3) implique (5.2) puisque peu importe x, y et t, au plus un q peut satisfaire $\phi_c(p)\!\downarrow = q$). On peut encore récrire (5.3) comme

$$(\forall x,y,t)(\exists q,t')[(t' \geq t)\ \&\ T(c,p,q,t')\ \&\ [T(p,x,y,t') \iff T(i,\langle q,x\rangle,y,t')]]$$

et donc, (5.1) comme

$$\begin{aligned}(\exists c)(\forall p,x,y,t)(\exists q,t')[(t' \geq t)\ \&\\ T(c,p,q,t')\ \&\ [T(p,x,y,t') \Leftrightarrow T(i,\langle q,x\rangle,y,t')]].\end{aligned}$$

□

5.7 Théorème *IA est Σ^0_3-ardu.*

Preuve. Rogers a montré que $COF \stackrel{\mathrm{d}}{=} \{x \mid W_x \text{ est co-fini}\}$ est Σ^0_3-complet [Rog87, Corollaire 14.XVI]. Nous donnons une réduction de COF à IA. Soit z un entier donné. Nous décrivons un algorithme, dépendant uniformément de z, pour une fonction partielle récursive. Cette fonction partielle récursive s'avérera la fonction universelle d'un SP acceptable ssi $z \in COF$. Nous aurons donc déterminé une fonction récursive f telle que pour tout z, $z \in COF \iff f(z) \in IA$, et le théorème sera établi. Le symbole ψ dénote un SP *acceptable* quelconque (e.g., $\psi = \phi$ suffit).

Voici l'algorithme :

Entrée : $\langle p,x\rangle$
Algorithme pour $\phi_{f(z)}$ **:**

1. Pour chaque y tel que $p \leq y \leq p + x$, calculer (sans limite sur le nombre d'étapes) $\phi_z(y)$. Si et lorsque tous ces calculs se terminent, passer à l'étape suivante.
2. Retourner $\psi_p(x)$.

□ **Algorithme**

Nous voulons montrer que $f(z) \in IA$ ssi $z \in COF$, c'est-à-dire que $\phi^{f(z)} \in \mathcal{SPA}$ ssi $z \in COF$.

5.8 Assertion Si W_z est co-infini, alors pour tout p, $\phi_p^{f(z)}$ est une fonction finie.

Preuve. Fixons p. Si W_z est co-infini, alors il existe $p' \geq p$ tel que $p' \notin W_z$, i.e., $\phi_z(p')\uparrow$. Par l'algorithme pour $\phi^{f(z)}$, alors pour tout $x \geq p' - p$, $\phi_p^{f(z)}(x)\uparrow$.

□ **Assertion 5.8**

Donc, clairement, si W_z est co-infini, $f(z) \notin IM$, et donc, $f(z) \notin IA \subseteq IM$. Par un raisonnement dual à celui de la preuve de l'Assertion 5.8, on montre que si W_z est co-fini, alors, $\phi_p^{f(z)} = \psi_p$ pour presque tous les p. Par la Proposition 1.14, donc, si W_z est co-fini, $\phi^{f(z)} \in \mathcal{SPA}$. Nous avons donc bien $f(z) \in IA \iff z \in COF$. □ **Théorème 5.7**

5.9 Corollaire *IA est Σ_3^0-complet.*

Comme corollaires à la preuve du Théorème 5.7, nous avons :

5.10 Corollaire (a) *IM est Σ_3^0-ardu.*
(b) *IMNA est Σ_3^0-ardu.*

Preuve. (a) : Immédiat par le fait que $IA \subseteq IM$ et que dans la preuve du théorème, si $z \notin COF$, alors $f(z) \notin IM$.

(b) : Dans la preuve de théorème, utilisons le SP ψ de la preuve du Théorème 1.12. Par la Proposition 1.13, toute variante finie de ψ est un SP exécutable maximal non-acceptable. Donc, si $z \in COF$, alors $f(z) \in IMNA$.

D'autre part, comme $IMNA \subseteq IM$, et comme on a toujours $z \notin COF \Longrightarrow \phi^{f(z)} \notin IM$, si $z \notin COF$, alors $f(z) \notin IMNA$. □

Nous concluons cette section en montrant que *IA* et *IMNA* sont conjointement affligés d'une forme particulièrement sévère d'inséparabilité. Ce résultat se rapporte à la question intuitive posée au début de la section à savoir s'il est possible en général de déterminer si un ordinateur possède un compilateur PASCAL, *sous l'hypothèse qu'on ne nous soumettra que des ordinateurs pouvant calculer toutes les fonctions partielles récursives*. S'il existait un ensemble récursif E tel que $IA \subseteq E$ et $IMNA \subseteq \overline{E}$, alors la question recevrait une réponse positive : en effet, pour déterminer si un interprète *maximal* soumis est acceptable ou non, il suffirait de déterminer s'il appartient à E, un problème décidable. On dirait alors que *IA* et *IMNA* sont *récursivement séparables*. S'il existait un *E récursivement énumérable* tel que $IA \subseteq E$ et $IMNA \subseteq \overline{E}$ (respectivement, $IMNA \subseteq E$ et $IA \subseteq \overline{E}$), la réponse serait "semi-positive", et on dirait que *IA* est *r.é. séparable de IMNA* (respectivement, *IMNA* est r.é. séparable de *IA*) [Case].

Case [Case] et Royer et Case [RoyCas] introduisent diverses notions d'inséparabilité *effective*, correspondant à divers degrés de sévérité dans la façon dont deux ensembles peuvent ne pas être récursivement séparables. Une des formes les plus sévères d'inséparabilité dont ils font mention (et une des plus sévères qui soient, indépendamment de la source) est la Σ_2^0-*inséparabilité effective*. Cette forme d'inséparabilité renforce de plusieurs façons la notion plus courante d'inséparabilité effective de Smullyan [Case, Rog87].

Intuitivement, A est effectivement Σ_2^0-inséparable de B (disjoint de A) ssi non seulement il n'y a aucun ensemble r.é. *dans le problème d'arrêt* E tel que $A \subseteq E$ et $B \subseteq \overline{E}$, mais encore, étant donné n'importe quel x, on peut *uniformément* produire un contrexemple à l'énoncé "$A \subseteq W_x^K$ et $B \subseteq \overline{W_x^K}$". Il s'agit donc d'une forme extrême d'inséparabilité. En voici la définition formelle.

5.11 Définition (Case [RoyCas]) *A est effectivement Σ_2^0-inséparable de B ssi $A \cap B = \emptyset$ et $(\exists f$ récursive$)(\forall x)[f(x) \in (W_x^K \cap B) \cup (A - W_x^K)]$.*

Nous montrons que *IA* et *IMNA* sont mutuellement inséparables dans ce sens extrêmement fort.

5.12 Théorème *IMNA est effectivement Σ_2^0-inséparable de IA.*

Preuve. Soit g une fonction récursive telle que pour tout i, $W_i^K = \{z \mid \lim_{k\to\infty} g(i,z,k) \text{ existe}\}$ (Fait 5.3). Soit f la fonction récursive (basée sur 1-pkrt), telle que pour tout x, $f(x)$ est un ϕ-programme réalisant l'algorithme suivant :

> **Entrée :** $\langle\langle p,n\rangle, z\rangle$
> **Algorithme pour** $\phi_{f(x)}$ **:**
> 1. Si pour tout $m \leq z$, $g(x, f(x), p+m) = g(x, f(x), p)$, alors retourner $\phi_p(z)$.
> 2. Autrement, calculer $\phi_p(n)$; ssi ce calcul s'arrête, retourner $\phi_p(z)$.
>
> □ **Algorithme**

Alors, nous avons le lemme suivant.

5.13 Lemme *Pour tout x, $f(x) \in W_x^K \Rightarrow f(x) \in IA$ et $f(x) \notin W_x^K \Rightarrow f(x) \in IMNA$.*

Preuve. Supposons que $e \stackrel{\mathrm{d}}{=} f(x) \in W_x^K$. Alors, il existe un p_0 tel que pour tout $p \geq p_0$, $g(x,e,p) = g(x,e,p_0)$. Alors, pour tout $p \geq p_0$, tout n et tout z, le test de l'Étape 1 de l'algorithme est satisfait, et nous avons $\phi^e_{\langle p,n\rangle}(z) = \phi_p(z)$. Ainsi donc, ϕ^e est programmable via la fonction

$$\lambda p.\langle rem(p, (\mu k)[rem(p,k) \geq p_0]), 0\rangle,$$

où rem est une instance effective de remb-inf dans ϕ, et donc, $e \in IA$.

Supposons maintenant que $e \notin W_x^K$. Alors, pour tout p, il existe un $q > p$ tel que $g(x,e,q) \neq g(x,e,p)$. Ainsi, pour tout p et n, il existe un z_0 tel que pour tout $z \geq z_0$, le test de l'Étape 1 de l'algorithme est insatisfait, et donc $\phi^e_{\langle p,n\rangle}(z)\downarrow = \phi_p(z) \Leftrightarrow \phi_p(n)\downarrow$. Autrement dit, pour tout p et n, si $\phi^e_{\langle p,n\rangle}$ n'est pas une fonction finie, alors $\phi_p(n)\downarrow$ et $\phi^e_{\langle p,n\rangle} = \phi_p$. Par un argument identique à celui dans notre preuve du Théorème 1.12, on montre alors que ϕ^e a puissance de calcul maximale, mais n'est pas programmable, autrement dit, que $e \in IMNA$. □ **Lemme 5.13**

Soit x quelconque. Si $f(x) \in W_x^K$, alors, par le lemme, $f(x) \in W_x^K \cap IA$.

Si $f(x) \notin W_x^K$, alors, également par le lemme, $f(x) \in IMNA - W_x^K$. Donc, $f(x) \in (W_x^K \cap IA) \cup (IMNA - W_x^K)$. □ **Théorème 5.12**

5.14 Théorème *IA est effectivement Σ_2^0-inséparable de IMNA.*

Preuve. Soit g comme dans la preuve précédente. Soit f la fonction récursive (basée sur 1-pkrt), telle que pour tout x, $f(x)$ est un ϕ-programme réalisant l'algorithme suivant :

> **Entrée :** $\langle\langle p, n\rangle, z\rangle$
> **Algorithme pour** $\phi_{f(x)}$ **:**
> 1. Si pour tout $m \leq n$, $g(x, f(x), p+m) = g(x, f(x), p)$, alors calculer $\phi_p(n)$; ssi ce calcul s'arrête, retourner $\phi_p(z)$.
> 2. Autrement, retourner $\phi_p(z)$.
>
> □ **Algorithme**

Nous avons le lemme suivant, dont découle le théorème, comme dans la preuve du théorème précédent.

5.15 Lemme *Pour tout x, $f(x) \in W_x^K \Rightarrow f(x) \in IMNA$ et $f(x) \notin W_x^K \Rightarrow f(x) \in IA$.*

Preuve. Supposons que $e \stackrel{\mathrm{d}}{=} f(x) \in W_x^K$ et soit p_0 tel que pour tout $p \geq p_0$, $g(x, e, p) = g(x, e, p_0)$. Nous montrons que pour chaque $p \leq p_0$, l'ensemble $\{\phi^e_{\langle p,n\rangle} \mid n \in N\}$ est fini, ce qui entraînera que $P \stackrel{\mathrm{d}}{=} \{\phi^e_{\langle p,n\rangle} \mid p \leq p_0 \;\&\; n \in N\}$ est fini.

Soit $p \leq p_0$. Si $(\forall p' > p)[g(x, e, p') = g(x, e, p)]$, alors clairement, quel que soit n, $\phi^e_{\langle p,n\rangle}$ ne peut être autre chose que ϕ_p ou $\lambda z.\!\uparrow$; si au contraire, il existe $n_0 > 0$ tel que $g(x, e, p) \neq g(x, e, p+n_0)$, alors pour tout $n \geq n_0$, $\phi^e_{\langle p,n\rangle}$ est une seule et unique fonction partielle, en l'occurrence, ϕ_p. Donc, P est fini.

D'autre part, par hypothèse sur p_0, pour tout $p \geq p_0$ et tout n, le premier test de l'Étape 1 de l'algorithme est satisfait. On a donc

$$(\forall p \geq p_0)(\forall n)[\phi^e_{\langle p,n\rangle} \neq \lambda z.\!\uparrow \Longrightarrow \phi^e_{\langle p,n\rangle}(n)\!\downarrow]. \tag{5.4}$$

On montre par un argument similaire à celui dans la preuve du Théorème 1.12 que ϕ^e a puissance de calcul maximale. Supposons maintenant

qu'il soit programmable, et soit f une fonction de programmation pour lui. Soit h une fonction récursive (basée sur 1-pkrt) telle que pour tout k,

$$\phi_{h(k)} = \lambda z. \begin{cases} \uparrow & \text{si } z = \pi_2(f(h(k))), \\ k & \text{autrement,} \end{cases}$$

où π_2 est la seconde fonction de projection de $\langle \cdot, \cdot \rangle$.

On note que pour tout k, k', $\phi_{h(k)} \neq \lambda z.\uparrow$, et que $k \neq k' \Rightarrow \phi_{h(k)} \neq \phi_{h(k')}$. Il existe donc k_0 tel que $\phi_{h(k_0)} \notin P$ (puisque cet ensemble, nous avons montré, est fini). Soient p et n tels que $\langle p, n \rangle = f(h(k_0))$. Puisque f est une fonction de programmation pour ϕ^e, on déduit que $\phi^e_{\langle p,n \rangle} \notin P$ et donc, par la définition de P, que $p > p_0$. À cause de cela, et puisque $\phi^e_{\langle p,n \rangle} \neq \lambda z.\uparrow$, on déduit par (5.4) que $\phi^e_{\langle p,n \rangle}(n)\downarrow$. Or, puisque f est une fonction de programmation pour ϕ^e, nous avons que $\phi^e_{\langle p,n \rangle} = \phi_{h(k_0)}$. D'un autre côté, par la propriété supposée de h, on sait que $\phi_{h(k_0)}$ diverge sur $\pi_2(f(h(k_0))) = n$; une contradiction, dont on déduit que ϕ^e n'est pas programmable, et que $e \in \mathit{IMNA}$.

Supposons maintenant que $e \notin W^K_x$. Alors, pour tout p, il existe un n tel que $g(x, e, p+n) \neq g(x, e, p)$. On vérifie qu'alors ϕ^e est programmable via la fonction

$$\lambda p.\langle p, (\mu n)[g(x, e, p+n) \neq g(x, e, p)] \rangle,$$

et que donc, $e \in \mathit{IA}$. □ **Lemme 5.15**

□ **Théorème 5.14**

Nous n'avons pas étudié la question de savoir si les Théorèmes 5.12 et 5.14 sont des cas particuliers d'un fait plus général. Il serait certes intéressant de voir si le fait que *IA* et *IMNA* sont des ensembles d'indices ou Σ^0_3-ardus a quelque-chose à dire dans le fait que ces théorèmes sont valides.

5.2 Constructivité de RVE à partir de prog

Dans cette section, nous faisons remarquer que les relations $\mathsf{prog} \models_e \mathsf{rs}$ pour chaque $\mathsf{rs} \in$ CC-RVE (Corollaire 2.71) sont toutes constructives. En termes plus intuitifs, il est possible, pour toute $\mathsf{rs} \in$ CC-RVE, d'inférer *uniformément* une instance de rs dans n'importe quel SP acceptable, à partir d'un interprète

de ce SP acceptable, et d'une fonction de programmation pour lui. Plus précisément :

5.16 Théorème *Pour toute* $\mathsf{rs} \in \mathsf{CC\text{-}RVE}$, *il existe une fonction* récursive f *telle que pour tout* i *et* j, *si* $i \in IA$ *et* ϕ_j *est une fonction de programmation pour* ϕ^i, *alors* $\phi_{f(i,j)}$ *est une instance de* rs *dans* ϕ^i.

La preuve de ce théorème est très simple mais fastidieuse. Elle consiste à vérifier que toutes les étapes des preuves du Théorème 2.69 et de l'interrelation $\mathsf{prog} \models_e \mathsf{1\text{-}pkrt}$ sont constructives. Sans faire en détail cet exercice, nous démontrons, à titre exemplaire, la constructivité de l'interrelation $\mathsf{prog} \models_e \mathsf{s\text{-}1\text{-}1}$.

5.17 Proposition *Il existe une fonction récursive* g *telle que pour tout* i *et tout* j, *si* $i \in IA$ *et* ϕ_j *est une instance de* prog *dans* ϕ^i, *alors* $\phi_{g(i,j)}$ *est une instance de* $\mathsf{s\text{-}1\text{-}1}$ *dans* ϕ^i.

Preuve. Soient s et s' des instances effectives de respectivement $\mathsf{s\text{-}1\text{-}1}$ et $\mathsf{s\text{-}2\text{-}1}$ dans ϕ. Soit a un ϕ-programme pour la fonction $\lambda\langle i,j,p,n\rangle.[\phi_j(s(s(i,p),n))]$. Définissons $g \stackrel{\mathrm{d}}{=} \lambda i,j.[s'(a,i,j)]$. Fixons i et j tels que ϕ^i est acceptable et ϕ_j est une fonction de programmation pour ϕ^i. Alors, pour tout p et tout n :

$$\begin{array}{rcll}
\phi^i(\phi_{g(i,j)}(\langle p,n\rangle)) & = & \phi^i(\phi_{s'(a,i,j)}(\langle p,n\rangle)) & \text{par définition de } g, \\
& = & \phi^i(\phi_a(\langle i,j,p,n\rangle)) & \text{par définition de } \mathsf{s\text{-}2\text{-}1}, \\
& = & \phi^i(\phi_j(s(s(i,p),n))) & \text{par définition de } \phi_a, \\
& = & \phi(s(s(i,p),n)) & \text{puisque } \phi_j \text{ est une fonction} \\
& & & \text{de programmation pour } \phi^i, \\
& = & \lambda z.[\phi_{s(i,p)}(\langle n,z\rangle)] & \text{par définition de } \mathsf{s\text{-}1\text{-}1}, \\
& = & \lambda z.[\phi_i(\langle p,n,z\rangle)] & \text{par définition de } \mathsf{s\text{-}1\text{-}1}, \\
& = & \lambda z.[\phi^i_p(\langle n,z\rangle)] & \text{par changement de notation.}
\end{array}$$

Par définition de $\mathsf{s\text{-}1\text{-}1}$, on a donc que $\phi_{g(i,j)}$ calcule une (version unarisée d'une) instance de $\mathsf{s\text{-}1\text{-}1}$ dans ϕ^i. □

Donc, on peut inférer uniformément une instance de $\mathsf{s\text{-}1\text{-}1}$ pour un SP acceptable à partir d'un interprète et d'une fonction de programmation pour celui-ci. Le Théorème 5.16 énonce qu'il en est de même pour toutes les RS

dans CC-RVE. En fait, l'uniformité des interrelations prog $\models_e$ CC-RVE va un peu plus loin : on peut montrer qu'il existe une unique fonction récursive f telle que pour toute rs $= (\mathcal{R}, P) \in$ CC-RVE, si on fournit à f un interprète acceptable i, un algorithme j pour une fonction de programmation pour ϕ^i, et des "descriptions" d'un opérateur récursif déterminant rs et d'un prédicat sur les séquences cumulativement co-fini témoignant du fait que P est CC, alors f retourne un algorithme pour une instance (évidemment effective) de rs dans ϕ^i. La "description" pourrait être, pour l'opérateur récursif, un indice d'une énumération standard de tous les opérateurs récursifs [Roy87, § 1.2], et pour le prédicat cumulativement co-fini, un algorithme permettant de vérifier sa valeur de vérité sur les séquences encodées d'une façon quelconque.

Ce degré d'uniformité nous amène à conclure que, dans un certain sens, la connaissance d'une instance effective de prog pour un SP exécutable représente vraiment la "capacité de programmer" dans ce SP. Avec un interprète et une instance effective de prog comme seules informations sur un SP, on peut inférer uniformément une instance effective de n'importe quelle RS "raisonnable" (i.e., dans CC-RVE) dont on sait donner une description des contraintes sémantiques et textuelles. Nous croyons qu'une instance effective de prog mérite bien le nom de "fonction de programmation"

Une autre interprétation de ce degré d'uniformité est qu'un compilateur pour un langage standard constitue une information *nécessaire et suffisante* pour programmer dans un langage dont on connaît un interprète : nécessaire, parce que si l'on ne sait pas compiler les programmes d'un langage standard, alors une technique à la fois simple et fondamentale de programmation nous échappe ; suffisante parce que la connaissance d'un compilateur pour un langage standard nous permet d'implanter une vaste classe d'autres techniques de programmation.

5.3 Difficulté d'inférer une fonction de programmation

Nous étudions maintenant la difficulté du problème général d'inférer une fonction de programmation à partir d'un interprète acceptable. Par "inférer une fonction de programmation", on veut dire calculer un ϕ-programme pour une telle fonction. Nous étudions ce problème comme un *problème avec promesse*, en ce sens que nous étudions sa solubilité par une fonction *par-*

tielle qui doit donner une bonne réponse si l'argument soumis est bien un interprète acceptable, mais qui peut avoir un comportement quelconque (en particulier, diverger) si ce n'est pas le cas. Comme pour chaque entrée valide possible il existe une infinité de bonnes réponses, nous pourrions dire qu'il s'agit d'un *problème de masse* [Rog87] *avec promesse* (cette notion n'a cependant jamais été définie formellement, à notre connaissance).

En vertu de la très grande puissance expressive *uniforme* de prog, soulignée en § 5.2, ce problème peut être vu comme le problème général d'apprendre à programmer dans un langage quelconque (mais garanti acceptable) sans autre connaissance du langage qu'un interprète de celui-ci, où "apprendre à programmer" signifie trouver en temps fini une "méthode générale de programmation", en l'occurrence, une fonction de programmation pour le langage en question. Nous montrons que *même avec un oracle dans le problème d'arrêt*, et donc même avec la possibilité de connaître magiquement l'issue de certaines exécutions impliquant des programmes du langage, toute procédure d'inférence d'une fonction de programmation à partir d'un interprète acceptable doit nécessairement se tromper "gravement".

Notons tout de suite qu'une interprétation intuitive de notre résultat est que la sémantique opérationnelle d'un langage (qui consiste, en dernière analyse, en la description d'un interprète du langage pour une certaine machine abstraite "standard" [Sto77, § 2]) ne constitue *pas* une description suffisante d'un langage pour les besoins d'un programmeur. Dans cette interprétation, notre résultat est, à première vue du moins, contraire à l'intuition et même à l'expérience. En effet, on considère habituellement que l'unique outil nécessaire et suffisant pour l'utilisation d'une machine est son manuel de "principes d'opération", qui décrit, essentiellement, la façon dont elle interprète son programme. Cette position est exprimée de façon très claire par Garwick [Gar66], qui ouvre son article avec la phrase "aucun langage de programmation ne peut être mieux défini que par son compilateur". Notre résultat montre que si cette position s'avère justifiée en pratique, c'est que les langages auxquels nous sommes exposés sont d'un type bien particulier, et que si *tous* les langages acceptables possibles étaient considérés, il faudrait obligatoirement dans certains cas avoir recours à un *guide du programmeur* pour apprendre à programmer dans un langage. Ce guide du programmeur devrait nécessairement contenir (explicitement ou implicitement) une procédure de compilation d'un langage standard vers le langage en question.

Voici plus précisément ce que dit notre résultat. Soit α une "procédure" (i.e.,

fonction) partielle récursive dans le problème d'arrêt. Intuitivement, α "prétend" pouvoir inférer une fonction de programmation pour un SP *acceptable* dès qu'on lui soumet un interprète de ce SP. Si on soumet autre chose qu'un interprète acceptable à α, alors elle peut faire n'importe quoi, y compris diverger. C'est pourquoi on dit que α est *partielle* récursive dans le problème d'arrêt. Alors, α ne "passe même pas près" d'être à la hauteur de ses prétentions en ce sens qu'il existe une *infinité* d'interprètes acceptables sur lesquels α, soit ne s'arrête même pas (i.e., diverge), soit retourne un ϕ-programme qui est "loin" de calculer une fonction de programmation pour le SP correspondant à l'interprète acceptable soumis. Par là, on veut dire que si t est la fonction (présumément totale) calculée par le ϕ-programme retourné par α et ψ le SP correspondant à l'interprète acceptable soumis à α, alors pour une infinité de p, soit $t(p)\uparrow$, soit $\psi_{t(p)}$ et ϕ_p diffèrent presque partout. (Si t était vraiment une fonction de programmation pour ψ, alors t serait définie partout, et pour tout p, il serait le cas que $\phi_p = \psi_{t(p)}$.)

5.18 Théorème *Pour toute fonction partielle récursive dans le problème d'arrêt α, il existe une infinité de $e \in IA$ tels que, soit $\alpha(e)\uparrow$, soit, posant $\psi \stackrel{\mathrm{d}}{=} \phi^e$ et $t \stackrel{\mathrm{d}}{=} \phi_{\alpha(e)}$, il existe une infinité de p pour lesquels soit $t(p)\uparrow$, soit ϕ_p et $\psi_{t(p)}$ diffèrent presque partout.*

Preuve. Soit α partielle récursive dans le problème d'arrêt, et soit g une fonction récursive telle que $\lambda x.[\lim_{y\to\infty} g(x,y)] = \alpha$ (Fait 5.4).

Soit rem une instance effective strictement croissante de remb dans ϕ. Soit k une fonction récursive telle que pour tout m, $k(m)$ est un ϕ-programme pour la fonction constante m, i.e., $\phi_{k(m)} = \lambda z.m$.

Pour tout u et n, on définit récursivement et conjointement A_u^n, B_u^n et c_u^n comme suit :

$$
\begin{array}{lcl}
A_u^0 & = & \emptyset, \\
c_u^0 & = & 0, \\
B_u^n & = & \begin{cases} \text{le singleton contenant le premier entier} \geq c_u^n \text{ obtenu en} \\ \text{effectuant pendant } n \text{ étapes une queue de colombe des} \\ \text{calculs } \phi_z(k(m)),\ m > 0, \text{ où } z = g(u,n), \text{ si un tel entier} \\ \text{est obtenu ; } \emptyset \text{ autrement,} \end{cases} \\
A_u^{n+1} & = & A_u^n \cup B_u^n, \\
c_u^{n+1} & = & \begin{cases} c_u^n & \text{si } B_u^n = \emptyset, \\ rem(b)+1,\ \text{où } \{b\} = B_u^n & \text{autrement.} \end{cases}
\end{array}
$$

On définit ensuite pour tout u

$$A_u \stackrel{\mathrm{d}}{=} \bigcup_{n \in N} A_u^n.$$

On note que pour tout u et n, A_u^n est un ensemble fini, peut-être vide. Il est clair que le prédicat $\lambda u, n, x.[x \in A_u^n]$ est récursif. Il est également clair que pour tout u, A_u est récursif, étant soit fini, soit énumérable en ordre strictement croissant (en appliquant la définition récursive). [1] On note aussi que pour tout u et pour tout p, $p \in A_u \Longrightarrow rem(p) \notin A_u$.

Soit maintenant a un ϕ-programme tel que

$$\phi_a = \lambda \langle u, p, x \rangle . \begin{cases} 0 & \text{si } p \in A_u^x \\ \phi_p(x) & \text{autrement.} \end{cases}$$

Soit krt une instance injective de krt. Alors, $krt(rem^j(a)),\ j \in N$ constitue une infinité de ϕ-programmes distincts. Soit e n'importe lequel de ces programmes. Nous montrons que $\psi \stackrel{\mathrm{d}}{=} \phi^e$ est acceptable et que $\alpha(e)$ soit est indéfini, soit est un ϕ-programme pour une fonction partielle (mais peut-être totale) t telle que soit $t(k(m))$ diverge pour presque tous les m, soit $\psi_{t(k(m))}$ est une variante finie de $\lambda x.0$ pour une infinité de $m > 0$. Comme pour tout m, $\phi_{k(m)} = \lambda x.m$, ceci établira le théorème.

1. Notons cependant qu'on ne peut ni décider si A_u est fini, ni résoudre récursivement le problème d'appartenance à A_u *uniformément en u*.

On note tout d'abord que, par définition de krt,

$$\phi_e = \lambda\langle p, x\rangle. \begin{cases} 0 & \text{si } p \in A_e^x \\ \phi_p(x) & \text{autrement.} \end{cases} \tag{5.5}$$

Définissons donc

$$f \stackrel{\mathrm{d}}{=} \lambda p. \begin{cases} rem(p) & \text{si } p \in A_e \\ p & \text{autrement.} \end{cases}$$

Puisque A_e est récursif, f est récursive. De plus, par le fait que pour tout p, $p \in A_e \Longrightarrow rem(p) \notin A_e$, on a que $\mathbf{image}(f) \cap A_e = \emptyset$. En particulier, pour tout p et pour tout x, $f(p) \notin A_e^x$. Par (5.5), on déduit que pour tout p, $\psi_{f(p)} = \phi_p$, c'est-à-dire que ψ est programmable via f, et donc acceptable.

Si $\alpha(e)\uparrow$, nous avons terminé ; supposons donc que $\alpha(e)\downarrow = z$, et soit n_0 tel que pour tout $n \geq n_0$, $g(e, n) = z$. Soient également $t \stackrel{\mathrm{d}}{=} \phi_z$ et $B \stackrel{\mathrm{d}}{=} \{t(k(m)) \mid m > 0 \;\&\; t(k(m))\downarrow\}$. Si B est fini, c'est que $t(k(m))\uparrow$ pour presque tous les m, et nous avons terminé. Soit donc B infini. Clairement, alors, par la définition des A_u^n, A_e est infini. De plus, $C \stackrel{\mathrm{d}}{=} A_e - A_e^{n_0} \subseteq B$. Puisque $A_e^{n_0}$ est fini, C est infini. Comme $C \subseteq A_e$, et par (5.5), pour chaque $c \in C$, ψ_c est une variante finie de $\lambda x.0$. Par contre, comme $C \subseteq B$, chaque $c \in C$ est l'image par t d'un ϕ-programme $k(m)$ pour un certain $m > 0$. Il existe donc une infinité de $m > 0$ pour lesquels $\psi_{t(k(m))}$ est une variante finie de $\lambda x.0$. $\square$

5.4 Non-constructivité de s-1-1 $\models_{em}$ prog

Nous montrons maintenant que la garantie de prog par s-1-1 dans les SP exécutables maximaux est non-constructive, en ce sens que le problème (qui pourrait lui aussi être décrit comme un problème de masse avec promesse) d'inférer une fonction de programmation pour un SP (garanti acceptable) à partir d'un interprète et d'une instance effective de s-1-1 pour ce SP est insoluble récursivement. Plus précisément, nous montrons que pour toute fonction 2-aire partielle récursive α, il existe une infinité d'interprètes acceptables a et de ϕ-programmes b tels que ϕ_b est une instance de s-1-1 pour ϕ^a, mais pour lesquels $\alpha(a, b)$, soit est indéfini, soit n'est *pas* un ϕ-programme calculant une fonction de programmation pour ϕ^a.

Notons que la garantie de prog par s-1-1 n'est en apparence que "légèrement non-constructive", puisque pour pouvoir construire une instance effective de

prog à partir d'une instance effective de s-1-1, il “suffit” de connaître un seul programme pour une fonction partielle bien précise, en l'occurrence, la fonction universelle (unaire) de ϕ. (En effet, pour tout ψ, si $\psi_u = \widehat{\phi}$ et s est une instance de s-1-1 en ψ, alors $\lambda p.s(u,p)$ est une instance de prog en ψ.)

5.19 Théorème *Pour toute fonction partielle récursive α, il existe une infinité de (e,s) tels que $e \in IA$ et ϕ_s est une instance de* s-1-1 *dans ϕ^e, et tels que $\alpha(e,s)$, soit est indéfini, soit n'est* pas *un ϕ-programme calculant une instance de* prog *dans ϕ^e.*

Preuve. Soit *remb* une instance effective, injective et strictement croissante en chaque argument de remb-inf dans ϕ. Soit s une instance effective, injective et strictement croissante en chaque argument de s-1-1 dans ϕ. On définit

$$sub \stackrel{\mathrm{d}}{=} \lambda p, n.remb(s(p,n),1).$$

On vérifie que *sub* est une instance effective, injective et strictement croissante en chaque argument de s-1-1 dans ϕ. On a aussi $sub(p,n) > \max(p,n)$ pour tout p et n.

Pour tout $k > 0$, toutes paires $\langle p,x\rangle$, $\langle q,y\rangle$ et toute séquence finie σ de longueur k (§ 1.3.8), on écrit $\langle p,x\rangle \stackrel{\sigma}{\to} \langle q,y\rangle$ ssi $x = \langle \sigma_1,\ldots,\sigma_k,y\rangle$ *et* $q = sub(\cdots sub(sub(p,\sigma_1),\sigma_2),\cdots,\sigma_k)$. On écrit aussi $\langle p,x\rangle \stackrel{\nu}{\to} \langle p,x\rangle$, où ν est la séquence vide.

On écrit $\langle p,x\rangle \equiv \langle q,y\rangle$ ssi il existe une σ (peut-être vide) telle que $[\langle p,x\rangle \stackrel{\sigma}{\to} \langle q,y\rangle] \vee [\langle q,y\rangle \stackrel{\sigma}{\to} \langle p,x\rangle]$. On vérifie facilement que $\equiv$, comme relation sur N, est une relation d'équivalence. Elle est aussi récursive, puisque *sub* est strictement croissante en chaque argument. On dit que $\langle p,x\rangle$ est *équivalente* à $\langle q,y\rangle$ ssi $\langle p,x\rangle \equiv \langle q,y\rangle$.

On vérifie, par le fait que *sub* est injective et strictement croissante en chaque argument, que pour toute $\langle p,x\rangle$ et tout q, il existe au plus un y pour lequel $\langle q,y\rangle \equiv \langle p,x\rangle$. On vérifie également que pour tout SP ψ, *sub* est une instance de s-1-1 dans ψ ssi pour toutes $\langle p,x\rangle$ et $\langle q,y\rangle$ telles que $\langle p,x\rangle \equiv \langle q,y\rangle$, $\psi_p(x) = \psi_q(y)$. En particulier, si $\langle p,x\rangle \equiv \langle q,y\rangle$, alors $\phi_p(x) = \phi_q(y)$.

Soit α n'importe quelle fonction partielle récursive. Nous montrons maintenant que pour un certain couple (e,s_0) spécifique, ϕ^e est acceptable, ϕ_{s_0} est une instance de s-1-1 dans ϕ^e, mais malgré tout, $\alpha(e,s_0)$ ne donne pas

une fonction de programmation pour ϕ^e. Le fait qu'il existe une infinité de tels couples s'établit par pré-rembourrage de s_0 et d'un "ancêtre" de e (avant passage dans une instance *injective* de krt), comme dans notre preuve du Théorème 5.18.

Soient s_0 un ϕ-programme pour sub et z_0 un ϕ-programme pour $\lambda z.0$. Par krt dans ϕ, il existe un ϕ-programme e réalisant l'algorithme suivant :

Entrée : $\langle p, x\rangle$
Algorithme pour ϕ_e **:**
{ Les β_n et γ sont implantées par indices canoniques. }

1. $n \leftarrow 0$.
2. $\beta_n \leftarrow$ une énumération pendant n étapes de $\widehat{\phi}$.
3. Si $\beta_n(\langle p, x\rangle)\downarrow$, retourner $\beta_n(\langle p, x\rangle)$; arrêter.
4. Faire n étapes du calcul $\alpha(e, s_0)$, puis, si un résultat est obtenu, du calcul $\phi_{\alpha(e,s_0)}(z_0)$. Si un résultat est obtenu pour $\phi_{\alpha(e,s_0)}(z_0)$, appeler ce résultat p_0 et passer à l'Étape 6 ; autrement, passer à l'Étape 5.
5. $n \leftarrow n + 1$; aller à l'Étape 2.
6. Trouver x_0, le plus petit entier y tel que $\langle p_0, y\rangle$ n'est équivalente à aucune $\langle p', x'\rangle \in \mathbf{dom}(\beta_n)$.
7. Si $\langle p, x\rangle$ est équivalente à $\langle p_0, x_0\rangle$, retourner 1 ; arrêter.
8. Retourner $\phi_p(x)$.

□ **Algorithme**

Nous montrons maintenant que ϕ^e est acceptable, et que $\phi_{s_0} = sub$ est une instance de s-1-1 pour ϕ^e. Par hypothèse sur α, nous aurons alors que $\alpha(e, s_0)\downarrow$, et que $\phi_{\alpha(e,s_0)}$ est une instance de prog dans ϕ^e. Clairement, l'Étape 6 de l'algorithme sera atteinte sur presque toutes les entrées. Soient p_0 et x_0 tels que calculés aux Étapes 4 et 6 lorsque cette dernière est atteinte. Clairement, l'Étape 6 est atteinte sur entrée $\langle p_0, x_0\rangle$. Par les Étapes 6 et 7 de l'algorithme, $\phi^e_{p_0}(x_0) = 1$, et donc, $\phi^e_{p_0} \neq \lambda z.0 = \phi_{z_0}$, une contradiction du fait que $\phi_{\alpha(e,s_0)}$ est une instance de prog dans ϕ^e. Le théorème sera donc établi.

5.20 Assertion ϕ^e est acceptable.

Preuve. Si $\alpha(e, s_0)\uparrow$ ou bien $\phi_{\alpha(e,s_0)}(z_0)\uparrow$, alors $\phi^e = \phi$ et est clairement acceptable. Autrement, soient p_0 la valeur obtenue à l'Étape 4, et x_0 tel que trouvé

à l'Étape 6 lorsque celle-ci est atteinte. Clairement, les seuls programmes qui n'ont pas la même sémantique en ϕ^e et en ϕ sont les programmes p tels qu'il existe un x pour lequel $\langle p, x\rangle \equiv \langle p_0, x_0\rangle$. Ces programmes sont précisément ceux de l'ensemble

$$A \stackrel{\mathrm{d}}{=} \{p \mid (\exists x)[\langle p, x\rangle \equiv \langle p_0, x_0\rangle]\}.$$

Par les propriétés de sub, on montre facilement que A est récursif et qu'exactement un élément de A, son plus petit, n'est pas dans **image**(sub). Appelons a_0 le plus petit élément de A, et posons $A' \stackrel{\mathrm{d}}{=} A - \{a_0\}$. Nous avons $a_0 < a$ pour tout $a \in A'$, et $A' \subseteq$ **image**(sub). Notons que tous les programmes dans **image**(sub) sont de la forme $remb(q, 1)$ pour un certain q ; donc, tous les programmes dans A' sont aussi de cette forme. Nous montrons que pour tout $a \in A$, $remb(a, 2) \notin A$. D'abord, par injectivité de $remb$, aucun des programmes $remb(a, 2),\ a \in A$ ne peut être dans A'. D'autre part, par monotonicité de $remb$, on a $remb(a_0, 2) > a_0$, et donc, $remb(a, 2) > a_0$ pour tout $a \in A'$. Donc, aucun des programmes $remb(a, 2),\ a \in A$ ne peut égaler a_0, et donc, pour tout $a \in A$, $remb(a, 2) \notin A$.

Définissons

$$t \stackrel{\mathrm{d}}{=} \lambda p. \begin{cases} p & \text{si } p \notin A, \\ remb(p, 2) & \text{autrement.} \end{cases}$$

On vérifie facilement, à l'aide des observations précédentes, que t est une instance effective de prog dans ϕ^e. Donc, ϕ^e est acceptable.

□ **Assertion 5.20**

5.21 Assertion sub est une instance de s-1-1 dans ϕ^e.

Preuve. Si $\alpha(e, s_0)\uparrow$ ou bien $\phi_{\alpha(e,s_0)}(z_0)\uparrow$, alors $\phi^e = \phi$ et clairement, sub est une instance effective de s-1-1 dans ϕ^e. Supposons donc le contraire, c'est-à-dire que l'Étape 6 est atteinte sur presque toutes les entrées. Rappelons que pour tout SP ψ, sub est une instance de s-1-1 dans ψ ssi pour toutes $\langle p, x\rangle$ et $\langle q, y\rangle$ telles que $\langle p, x\rangle \equiv \langle q, y\rangle$, $\psi_p(x) = \psi_q(y)$. Soient p_0 la valeur obtenue à l'Étape 4, et x_0 tel que trouvé à l'Étape 6 lorsque celle-ci est atteinte. Clairement, les seules paires $\langle p, x\rangle$ pour lesquelles $\phi_p(x) \neq \phi^e_p(x)$ sont celles qui sont équivalentes à $\langle p_0, x_0\rangle$. Donc, pour toutes $\langle p, x\rangle$ et $\langle q, y\rangle$, si $\langle p, x\rangle \equiv \langle q, y\rangle \not\equiv \langle p_0, x_0\rangle$, alors $\phi^e_p(x) = \phi^e_q(y)$. D'autre part, si $\langle p, x\rangle \equiv$

$\langle q, y\rangle \equiv \langle p_0, x_0\rangle$, alors $\phi^e_p(x) = \phi^e_q(y) = \phi^e_{p_0}(x_0) = 1$. Donc, pour toutes $\langle p, x\rangle$ et $\langle q, y\rangle$ telles que $\langle p, x\rangle \equiv \langle q, y\rangle$, $\psi_p(x) = \psi_q(y)$, et donc, *sub* est une instance de s-1-1 dans ϕ^e. □ **Assertion 5.21**

□ **Théorème 5.19**

Nous croyons que l'énoncé du Théorème 5.19 est valide pour toute α partielle récursive dans le problème d'arrêt. Cependant, aucune extension évidente de notre technique de preuve ne permet d'obtenir ce résultat.

Notons que le Théorème 5.19 donne en corollaire une version "faible" (s'appliquant aux fonctions partielles récursives *sans oracle*) du Théorème 5.18. Pour poursuivre la discussion au sujet de ce qui constitue (ou plutôt, ne constitue *pas*) une description adéquate d'un langage de programmation (§ 5.3), le Théorème 5.19 nous dit qu'un interprète, *même accompagné d'un algorithme pour une instance de* s-1-1, ne constitue toujours pas une description suffisante d'un langage pour les besoins d'un programmeur.

Conclusion

Une des conclusions de ce travail qui s'imposent est que la RVE constitue une notion fructueuse pour l'étude des propriétés de programmation des SP. Elle permet l'expression d'une classe strictement plus grande de "rapports sémantiques" que la structure de contrôle de Riccardi, et le résultat de complétude expressive auquel elle donne lieu (le Théorème 2.69) est donc strictement plus informatif sur la similitude et la versatilité des SP acceptables, que celui de Case et Royer basé sur les structures de contrôle.

La RS constitue clairement aussi un cadre limitatif utile et naturel pour évaluer, comparer et relier différents formalismes pour l'expression de "rapports sémantiques". Nous l'avons utilisée entre autres pour étudier la RVE en relation avec la structure de contrôle. En particulier, l'extensionnalité (§ 2.5), définie dans le contexte général de la RS, s'avère correspondre à une notion similaire définie par Royer dans le cadre plus restreint des structures de contrôle.

En ce qui a trait à la complexité des instances effectives de RS, notre conclusion est que s-1-1 est strictement plus fondamentale que comp : La propriété d'avoir une instance "facile" de s-1-1 est beaucoup plus déterminante pour "l'utilité pratique" d'un SP que celle d'avoir une instance "facile" de comp. Cependant, la différence entre comp et s-1-1, de ce point de vue, est moindre que ne l'avaient soupçonné Machtey, Winklmann et Young [MWY78] et Royer [Roy87].

Le prédicat d'une RS (tout comme celui d'une structure de contrôle) représente une contrainte syntaxique (ou textuelle) sur les implantations de la technique de programmation correspondante. Jusqu'à quel point cette composante de la RS est-elle naturelle en termes de technique de programmation ? Certaines RS ont un prédicat intuitivement "trop" exigeant, malgré

qu'elles soient récursivement satisfaisables dans tout SP acceptable (Théorème 3.12). On ne peut exiger qu'un SP ait une instance effective "facile" d'une telle RS pour dire qu'il est "pratiquement utilisable". Cependant, nous avons montré que les membres de la classe LC-RVE, qui contient des RS imposant certaines contraintes syntaxiques, mais que l'on pourrait qualifier de "raisonnables", ont *toutes* une instance effective "facile" dans tout SP acceptable possédant une instance effective "facile" de s-1-1. On peut donc se permettre d'exiger d'un SP qu'il ait une instance effective facile de toutes les RS dans LC-RVE, *que l'on considère naturel ou non l'imposition de contraintes syntaxiques*. Ainsi, LC-RVE apparaît comme une grille de référence au moins "raisonnable" pour estimer le caractère pratique de la tâche de programmer dans un SP. Rappelons que LC-RVE contient les RS remb, remb-inf, remb-inj et remb-inf-inj, fréquemment utilisées, et qui ne tombaient pas sous le coup du théorème de complétude expressive de Case et Royer.

Au sujet de la non-constructivité des interrelations de garantie entre RS, nos résultats suggèrent que la connaissance d'une instance effective de prog constitue une information *nécessaire et suffisante* pour programmer dans un SP acceptable, laquelle n'est en général *pas* inférable à partir d'un interprète du SP acceptable utilisé. Ainsi, la description complète d'un interprète d'un langage de programmation (en d'autres mots, sa *sémantique opérationnelle* [Sto77, § 2]) ne serait pas, en général, une description suffisante d'un langage pour un programmeur ; une description de langage destinée à un programmeur devrait en général inclure, explicitement ou implicitement, la description d'un compilateur d'un langage standard vers le langage décrit.

Une *classe de Rogers* [Roy87, Définition 1.4.2.1] est une classe de la relation d'équivalence induite sur $\mathcal{SP}$ par la relation d'ordre partiel de traductibilité ($\psi \leq_R \psi' \overset{\mathrm{d}}{\iff} (\exists f \text{ réc.})(\forall p)[\psi_{f(p)} = \psi'_p]$). Deux membres d'une telle classe sont inter-traduisibles dans les deux directions. Royer a étudié la "transmission" de structures de contrôle d'un membre d'une classe de Rogers aux autres membres de la même classe [Roy87, § 2.3]. Entre autres, il a montré que les structures de contrôle extensionnelles (avec ou sans récursion) sont "transmises" d'un membre à l'autre d'une même classe de Rogers. Il serait certes intéressant de faire la même étude pour les RVE ; c'est un domaine que nous n'avons pas exploré. Outre dans nos théorèmes de complétude expressive (les Théorèmes 2.69 et 3.4), nous n'avons pas non plus étudié d'interrelations de garantie impliquant des RVE qui ne sont pas de Riccardi ; c'est aussi un sujet qui mériterait d'être abordé.

Sur l'aspect complexité des interrelations de garantie, notre Théorème 3.13 montre que tout SP possédant une instance effective dans $\mathcal{L}intime$ de comp possède une instance effective dans $\mathcal{P}time$ de s-1-1. Mais l'inverse est-il vrai ? Tout SP possédant une instance effective dans $\mathcal{P}time$ de s-1-1 possède-t-il une instance effective dans $\mathcal{L}intime$ de comp ? Nous soupçonnons que la réponse est négative. Rappelons que la classe des SP possédant une instance effective dans $\mathcal{P}time$ de s-1-1 correspond intuitivement, par notre Théorème 3.4, aux SP dont la "tâche de programmer" est "faisable" (sous l'hypothèse que les calculs "faisables" sont ceux que l'on peut exécuter en temps polynomial), et constitue donc une classe importante de SP.

En ce qui concerne les interrelations de garantie dans les SP maximaux, peut-on montrer qu'il n'existe aucune RS, mettons parmi les RVE à prédicat $VRAI$ (pour circonscrire la question), qui soit *strictement* plus faible que prog dans les SP maximaux, mais qui garantisse quand même l'acceptabilité dans les SP exécutables maximaux ? Nous soupçonnons qu'il n'existe aucune telle RS.

Concernant l'impossiblité d'inférer des instances effectives de RS, nous croyons que certains résultats spécifiques pour d'autres RS que prog et s-1-1 pourraient être obtenus en adaptant les preuves des Théorèmes 5.18 et 5.19. Mais peut-on obtenir des résultats généraux ? Existe-t-il une classe de RS pour lesquelles on peut montrer collectivement qu'il est en général impossible d'en inférer une instance effective dans un SP acceptable à partir d'un interprète de ce SP acceptable ? De même, existe-t-il une classe de relations sémantiques rs dont on peut montrer collectivement qu'il est en général impossible d'inférer une instance effective de prog pour un SP acceptable à partir d'un interprète de ce SP acceptable *et* d'une instance effective de rs dans ce même SP acceptable ?

À notre avis, une des avenues les plus attrayantes pour la poursuite des recherches exposées dans ce travail serait leur adaptation à des contextes sémantiques plus "réalistes" que la sémantique "simpliste" des SP. Une sémantique comme celle du langage "Reflect" de [HNTJ89], qui incorpore une forme d'auto-référence *intensionnelle* (non-exprimable en sémantique dénotationnelle), nous semblerait a priori un point de départ prometteur pour un tel exercice.

Bibliographie

[Alt80] D. Alton. “Natural” programming languages and complexity measures for subrecursive programming languages : an abstract approach. Dans *Recursion Theory : its Generalisations and Applications*, F. Drake et S. Wainer, éd., Cambridge University Press, 1980.

[BB88] G. Brassard et P. Bratley. *Algorithmics : Theory and Practice.* Prentice-Hall, 1988.

[Bir90] R. S. Bird. A Calculus of Functions for Program Derivation. Dans : *Research Topics in Functional Programming*, D. A. Turner, éd., Addison-Wesley, 1990.

[Blu67] M. Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14, 1967.

[BL74] W. Brainerd, L. Landweber. *Theory of Computation*. John Wiley and Sons, 1974.

[Case] J. Case, Effectivizing inseparability. À paraître dans *Z. Math. Logik Grundlagen Math.*

[CB71] R. Constable, A. Borodin. Subrecursive programming languages I : efficiency and program structure. *J. ACM*, 19 :526-568, 1971.

[Gar66] J. V. Garwick. The definition of programming languages by their compilers. Dans : Steel, T. B. (éd.) : Formal language description languages for computer programming. Proc. IFIP Working Conf. 1964. Amsterdam : North-Holland 1966, pp. 266-294.

[Gre75] S. Greibach. *Theory of Program Structure : Schemes, Semantics, Verification.* LNCS 36 (Springer-Verlag, 1975).

[Har74] J. Hartmanis. Computational complexity of formal translations. *Math. Systems Theory*, 8, 1974.

[Har82] J. Hartmanis, A note on natural complete sets and Gödel numberings, *Theor. Comput. Sci.* **17** (1982) 75–89.

[HB75] J. Hartmanis et T. Baker, On simple Gödel numberings and translations, *SIAM J. Comput.*, **4** (1975) 1–11.

[HNTJ89] T. Hansen, T. Nikolajsen, J. Träff et N. Jones. Experiments with implementations of two theoretical constructions. *Proc. Logic at Botik Conference 1989*, LNCS 363, Springer-Verlag.

[HU79] J. Hopcroft et J. D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, 1979).

[Jon90] N. D. Jones, Partial evaluation, self-application and types. Dans : *Proceedings, ICALP 90*, LNCS 443 (Springer-Verlag, Berlin, 1990).

[Kle38] S. Kleene, On notation for ordinal numbers. *J. Symbolic Logic*, 3 :150–155.

[Kur89] S. Kurtz, Communication personnelle (1989).

[LN85] O. Lecarme et J.-L. Nebut. *Pascal pour programmeurs*, McGraw-Hill, Paris, 1985.

[Marc] Y. Marcoux, Fully time-constructible real numbers. En préparation.

[Mar89] Y. Marcoux, Composition is Almost as Good as S-1-1. Dans : *Proceedings of the Fourth Annual Conference on Structure in Complexity Theory*, IEEE Computer Society Press (1989).

[MS55] J. Myhill et J. Shepherdson, Effective operations on partial recursive functions. *Z. Math. Logik Grundlagen Math.* **1** (1955) 310–317.

[MWY78] M. Machtey, K. Winklmann and P. Young, Simple Gödel numberings, isomorphisms, and programming properties, *SIAM J. of Comput.* **7** (1978) 39–60.

[MY78] M. Machtey et P. Young, *An Introduction to the General Theory of Algorithms* (North-Holland, Amsterdam, 1978).

[MY81] M. Machtey et P. Young, Remarks on recursion versus diagonalization and exponentially difficult problems, *J. Comput. Systems Sci.* **22** (1981) 442–453.

[Ric80] G. Riccardi, The independence of control structures in abstract programming systems, Ph.D. Thesis, State University of New York at Buffalo, 1980.

[Ric81] G. Riccardi, The independence of control structures in abstract programming systems, *J. Comput. Systems Sci.* **22** (1981) 107–143.

[Ric82] G. Riccardi, The independence of control structures in programmable numberings of the partial recursive functions, *Z. Math. Logik Grundlagen Math.* **48** (1982) 285–296.

[Rog58] H. Rogers, Gödel numberings of the partial recursive functions, *J. Symbolic Logic* **23** (1958) 331–341.

[Rog67] Édition originelle de [Rog87] (McGraw-Hill, 1967).

[Rog87] H. Rogers, *Theory of Recursive Functions and Effective Computability* (MIT Press, 1987).

[Roy87] J. Royer, *A Connotational Theory of Program Structure*, LNCS 273 (Springer-Verlag, 1987).

[RC86] J. Royer et J. Case, Progressions of relatively succinct programs in subrecursive hierarchies, Technical Report 86-007, Computer Science Department, University of Chicago, 1986.

[RoyCas] J. Royer et J. Case, *Intensional Subrecursion and Complexity Theory*, Research Notes in Theoretical Computer Science (Pitman Press, en révision).

[Sto77] J. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[You88] P. Young. Juris Hartmanis : fundamental contributions to isomorphism problems, Technical Report 88-06-02, Computer Science Department, University of Washington, 1988.

Index des symboles

$\stackrel{\mathrm{d}}{=}$: § 1.3.1
N : § 1.3.1
$\underline{n}$: § 1.3.1
2^A : § 1.3.1
$|x|$: § 1.3.1
$|(x_1, \ldots, x_n)|$: § 1.3.1
$|x_1, \ldots, x_n|$: § 1.3.1
lg : § 1.3.1
μ : § 1.3.2
dom : § 1.3.2
image : § 1.3.2
$\downarrow$: § 1.3.2
$\uparrow$: § 1.3.2
$\lambda x_1, \ldots, x_n.[\cdots]$: § 1.3.2
$\circ$: § 1.3.2
$\oplus$: § 1.3.2
$O, \ \Omega$: § 1.3.3
$\leftarrow$: § 1.3.4
c_A : § 1.3.5
$\langle \cdot, \cdot \rangle$: § 1.3.5
$\lambda \langle x_1, \ldots, x_n \rangle.[\cdots]$: § 1.3.5
$\Sigma_k, \ \Pi_k$: § 1.3.5
$\rho, \ \psi$: § 1.3.6
M_i : § 1.3.6
ϕ : p. 25
$\psi(p), \ \phi(p)$: § 1.3.6
$\psi_p, \ \phi_p$: § 1.3.6
$\widehat{\phi}, \ \widehat{\psi}$: § 1.3.6
W_i : § 1.3.7
K : § 1.3.7
$\phi_i^K, \ W_i^K$: § 1.3.7
D_u : § 1.3.7
Φ_i : § 1.3.7
$\Gamma, \ \Theta, \ \Phi, \ \Psi$: § 1.3.7
$\sigma, \ \ell(\sigma)$: § 1.3.8
ν : § 1.3.8
$x : y$: § 1.3.8
$\mathcal{R}$, rs : p. 40
$\models, \ \equiv\!\!\models, \ \not\models, \ \not\equiv\!\!\models$: p. 70
IA, IM, $IMNA$: p. 139
ϕ^e : p. 139

Index des définitions

acc : p. 51
acceptable (SP) : p. 26
application : § 1.3.2
arité : § 1.3.1, § 1.3.2
bijection : § 1.3.2
CC (prédicat) : p. 74
CC-RVE : p. 76
comp : p. 49
couple : § 1.3.1
ElmRec : § 1.3.4
exécutable (SP) : p. 25
Exptime : § 1.3.4
fonction : § 1.3.2
fonction caractéristique : § 1.3.5
fonction partielle : § 1.3.2
garantie (relations de) : § 2.6, p. 70
hyperimmunisé (ensemble) : § 1.3.5
immunisé (ensemble) : § 1.3.5
indice canonique : § 1.3.7
injection : § 1.3.2
instance : p. 34, p. 39
krt : p. 50
left-comp$_\alpha$: p. 120
Lintime : § 1.3.4
majoration : p. 48
LC (prédicat) : p. 80
LCR (prédicat) : p. 85
LCR-RVE : p. 85
LC-RVE : p. 80
maximal (SP) : § 1.3.6
mono-s-1-1 : p. 81
MR (prédicat) : p. 38
MR-RIC : p. 72
Myhill-Shepherdson (RS de) : p. 70
opérateur d'énumération : § 1.3.7
opérateur récursif : § 1.3.7
pairage (fonction de) : § 1.3.5
paire : § 1.3.5
Part : § 1.3.2
partielle récursive (fonction) : § 1.3.5
PartRec : § 1.3.5
PartRecUn : § 1.3.5
PartUn : § 1.3.2
path1, path2 : p. 78
n-pkrt : p. 50
po-comp-h : p. 118
PORT : p. 72
portable (RS) : p. 72
PrimRec : § 1.3.4
prog : p. 26, p. 49
programmable (SP) : p. 26
programmation (fonction de) : p. 26
programme : § 1.3.6
projection (fonction de) : § 1.3.5
propriété de programmation : § 1.1
pr-comp-h : p. 118
pseudo-inverse : p. 38
ps-inv : p. 42

$\mathcal{P}time$: § 1.3.4
purement extensionnelle (RS) : p. 52
queue de colombe (*"dovetail"*) : § 1.3.4
rech-nb : p. 48
récursif (ensemble) : § 1.3.5
récursive (fonction, relation) : § 1.3.5
récursivement énumérable (ensemble) : § 1.3.5
relation : § 1.3.2
relation sémantique : p. 39
relation vérifiable par énumération : p. 43
remb : p. 50
remb-inf : p. 50
RIC : p. 45
Riccardi (RS de) : p. 41, p. 45
right-comp$_\alpha$: p. 120
Royer (RS extensionnelle au sens de) : p. 53
RS : voir relation sémantique
RVE : voir relation vérifiable par énumération
RVE : p. 45
séquence d'entiers : § 1.3.8
semi-effective (instance) : p. 120
SE1, SE2 : p. 120
SP : voir système de programmation
$\mathcal{SP}$, $\mathcal{SPE}$, $\mathcal{SPM}$, $\mathcal{SPEM}$, $\mathcal{SPP}$, $\mathcal{SPA}$: p. 27
strange : § 4.4
structure de contrôle : p. 34
surjection : § 1.3.2
système de programmation : p. 24
s-1-1, s-m-1 : p. 49
s-1-1-cr : p. 85
$\mathcal{T}ot$: § 1.3.2
totale (fonction partielle) : § 1.3.2
tuplet : § 1.3.1
unarisée (version) : § 1.3.5
univ : p. 73
univaluée (relation) : § 1.3.2
univaluée (RS) : p. 52
universelle (fonction) : § 1.3.6
UTIL : p. 70
variante finie : § 1.3.2
$VRAI$: p. 37
VRAI-RIC : p. 80