THÈSE

présentée à

L'UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
École Doctorale - Sciences pour l'Ingénieur

pour obtenir le titre de

Docteur en Sciences - Spécialité Informatique

par

Renaud MARLET

Titre de la thèse :

VERS UNE FORMALISATION DE L'ÉVALUATION PARTIELLE

Soutenue le 15 décembre 1994 à Sophia Antipolis devant le jury composé de :

MM.	**Laurent KOTT**	Président et rapporteur
	Jean-Claude RAOULT	Rapporteur
	Paul FRANCHI-ZANNETTACCI	Directeur
	Gilles KAHN	Examinateurs
	Philippe DEVIENNE	
	Jacques CHAZARAIN	

aux Pénélopes

Vers une Formalisation de l'Évaluation Partielle

Résumé.

Cette thèse propose une formalisation de l'évaluation partielle. Elle fournit par ailleurs quelques outils d'aide à la construction d'évaluateurs partiels (correction des transformations, algorithme, terminaison). Elle est motivée par le constat de lacunes dans des fondements théoriques et dans des mises en œuvre pratiques. L'une, notamment, tient au caractère informel de la problématique de l'évaluation partielle : « déterminer des programmes qui sont à la fois équivalents et plus performants qu'un programme donné ».

Afin de modéliser précisément la performance d'un programme, nous proposons tout d'abord un cadre général qui permet de représenter le processus d'exécution (par opposition à la sémantique pure). Nous indiquons ensuite comment attribuer un coût à une exécution et étudions plusieurs mesures d'efficacité. Cette présentation débouche sur une définition formelle de l'évaluation partielle en termes d'optimisation. Elle met également en évidence plusieurs notions d'évaluation partielle optimale. Cette approche est illustrée d'exemples qui reposent sur la sémantique naturelle et les schémas de programmes récursifs.

Nous employons ces mêmes schémas pour étudier formellement la correction des transformations de pliage/dépliage, transformations majeures de l'évaluation partielle. En particulier, nous établissons quelques lemmes de commutativité sur les systèmes de réécriture afin d'étendre le champ d'application des conditions de correction du dépliage/pliage faible. D'autre part, nous donnons une définition formelle à l'algorithme de redémarrage généralisé, montrons rigoureusement sa terminaison, et examinons quels bons critères d'arrêt peuvent l'équiper.

Nous concluons notre étude sur une analyse des limites à l'optimisation par évaluation partielle.

Mots-clés : évaluation partielle, mesure de performance, optimalité, schémas de programme, pliage/dépliage, réécriture, redémarrage généralisé, terminaison.

Towards a Formalization of Partial Evaluation

Abstract.

This thesis proposes a formalization of partial evaluation. Moreover, it provides some tools for the construction of a partial evaluator (with discussions of correction of transformations, algorithms, and termination). It is motivated by deficiencies in some theoretical foundations and some practical implementations. One notable point relates to the informal status of the partial evaluation problem: "to determine programs which are both equivalent and more efficient than a given program."

In order to precisely model the performance of a program, we first give a general setting for representing the process of execution. We then show how to assign a cost to an execution and study several measurements of efficiency. This presentation leads to a formal definition of partial evaluation in optimization terms. It also brings out several notions of optimal partial evaluation. It is illustrated with examples built on structural operational semantics and recursive program schemes.

Those schemes are also used to formally study the correction of fold/unfold transformations, the main transformations of partial evaluation. In particular, we state some lemmas about commutativity and rewriting in order to extend the applications of correction conditions for weak unfold/fold transformations. Moreover, we give a formal definition to the generalized restart algorithm, show rigorously its termination, and investigate which good termination criterion it may use.

As a conclusion, we analyze the limits of optimization by partial evaluation.

Keywords : partial evaluation, performance measurement, optimality, program schemes, fold/unfold, rewriting, generalized restart, termination.

Table des Matières

Avant-propos

Le théatre.

Grâce aux développements de l'ingénierie logicielle et aux progrès rapides de l'électronique, indispensable support matériel, l'informatique s'est imposée comme technologie. Elle n'a cependant pu exister en tant que science qu'une fois établies les bases rigoureuses de la calculabilité[1].

À l'orée du siècle, le mathématicien David Hilbert lance un programme de recherche afin de trouver une méthode générale pour déterminer si une proposition mathématique quelconque est vraie ou fausse. L'ambition est grande, et le sujet, fertile. Mais la réponse tombe en 1931, implacable. Kurt Gödel démontre par son théorème d'incomplétude qu'une telle méthode ne peut exister. On cherche alors à cerner le concept, encore imprécis, de fonction calculable. Ainsi naissent dans les années 30 les systèmes équationnels de Herbrand-Gödel, le λ-calcul de Church, les fonctions partielles récursives de Kleene, la machine de Turing et les systèmes de Post. La thèse de Church va ensuite démontrer que toutes ces formulations sont équivalentes. Aujourd'hui encore, tout ce que l'on sait réellement calculer tient là.

« Définir n'est pas calculer » est la leçon fondamentale de Gödel. En particulier, il existe des problèmes dont on sait pertinemment qu'ils ont des solutions, mais qu'aucune construction ne permet pourtant d'atteindre[2]. Il faut donc clairement distinguer une définition constructive, qui repose sur un procédé effectif, d'une définition non-constructive, qui se contente de décrire sans donner de moyen explicite d'accéder aux objets.

Vues sous l'angle de la programmation, les fonctions calculables sont la description de séquences d'opérations élementaires qui transforment un ensemble de données en un résultat. Une difficulté majeure complique leur étude et fait aussi leur richesse : on ne sait généralement pas garantir que ces séquences ont bien une fin. Même si l'on reste en deçà de cette limite de décidabilité, certaines méthodes, bien que mathématiquement correctes, sont impraticables car elles mettent en jeu des calculs astronomiques à l'échelle de l'homme.

La pièce.

C'est pourquoi, aujourd'hui encore, la programmation reste un « art ». Pourtant, l'essor de l'informatique dans des secteurs critiques demande une rigueur nouvelle, qui ne se satisfait pas de recettes et d'incertitudes.

L'évaluation partielle se pose le problème à deux niveaux. Tout d'abord, elle prend un programme (supposons qu'il fonctionne) et en fabrique un autre qui lui est équivalent. Ensuite, elle prétend que ce second programme est meilleur que le précédent.

[1] Cet avant-propos est inspiré de textes moissonnés dans [Gri91, HU79, BDG88, Sav82, Liv78, Lee90].

[2] On peut en avoir une idée intuitive en réalisant qu'il existe beaucoup plus de fonctions que de fonctions calculables. En effet, l'ensemble des fonctions calculables, par définition consitué de fonctions décrites par un énoncé fini, est nécessairement dénombrable. Il ne pourra donc jamais contenir un ensemble aussi insignifiant que celui des fonctions de $\mathbb{N}$ dans $\{0,1\}$, pourtant comparable à $\mathbb{R}$.

Le premier point est crucial. Si l'on ne peut pas garantir que le nouveau programme est équivalent, il est extrêmement dangereux de le substituer à l'ancien car, non seulement les nouveaux calculs peuvent être erronés, mais de plus, rien ne signale un possible comportement différent.

Le second, a priori simple question de confort, est en fait riche en conséquences. Il ne veut pas seulement dire que le programme va donner ses résultats plus rapidement, ou en consommant moins de ressources ; lorsque l'on peut garantir le gain de performance, il encourage aussi les programmeurs à penser en termes de spécifications plutôt que d'implémentations, et à écrire du code plus naturel avec moins de préoccupations d'optimisation en tête[3]. Par ailleurs, des calculs humainement trop longs, ou nécessitant une mémoire trop importante, peuvent devenir possibles.

L'enjeu est important. L'industriel est satisfait par une rentabilité accrue (programmes plus sûrs, écrits plus vite, plus efficaces et plus faciles à maintenir). Le programmeur est satisfait parce qu'il fait dans de meilleures conditions une tâche plus gratifiante car de plus haut niveau.

Évidemment, nous en sommes encore loin, et cette thèse, qui s'interroge sur la validité des garanties d'équivalence et d'efficacité, ne met qu'une touche hâtive au tableau avant de passer la main.

Les protagonistes.

Il n'aurait pas été humainement possible d'écrire ce document au nombreux signes cabalistiques sans TEX, LATEX et EMACS, chacun honteusement « hacké ».

La préparation physique et psychologique, outre les interventions avisées de ma famille et de mes amis, a été assurée par le café Lavazza « Expresso », l'aspirine tamponnée effervescente Oberlin, et les variations Goldberg de Jean-Sébastien Bach interprétées par Glenn Gould (1981).

On ne livre pas de guerre sans une bonne intendance. Je suis admiratif devant les moyens humains et matériels que j'ai trouvé à l'INRIA et à l'Université d'Édimbourg ; ils m'ont permis de poursuivre mes recherches dans des conditions exceptionnelles. J'ai également reçu l'aide de l'UNSA en la personne de Danielle Gérin pour l'organisation de la soutenance. Je remercie par ailleurs l'École polytechnique et l'INRIA, qui m'ont accordé une bourse afin que je puisse réaliser ce travail, ainsi que le LFCS et Simulog, qui m'ont donné les moyens matériels de l'achever après mon départ de Sophia.

Pour le soutien qu'ils m'ont apporté, notamment sur le plan scientifique et technique, je suis reconnaissant à Yves Bertot, Laurent Hascoet, André Hirschowitz, Ian Jacobs, Thierry Le Sergent, Kevin Mitchell, Francis Montagnac, Vincent Prunet, Christophe Raffalli, Benoit Rottembourg, Jean-Bernard Saint, Bruno Salvy, Jean-Marc Steyaert, Laurent Thery, Paul Zimmermann. À ceux-là, il faut également ajouter la voix désincarnée des « news », insondable puits d'informations, et les minettes du service de documentation, mariage de charme et d'efficacité.

Je remercie Jean-Claude Raoult pour sa lecture très (trop) attentive, ainsi que Laurent Kott, qui a accepté la double charge de rapporteur et de président du jury. Je remercie également Paul Franchi-Zannettacci, Philippe Devienne et Jacques Chazarain de s'être intéressés à ce travail.

Je rends hommage au projet CROAP, mon projet d'accueil à l'INRIA Sophia Antipolis, pour ses bonnes attentions et pour son courage à supporter stoïquement mes facéties.

[3] C'est notamment l'idée contenue dans l'élimination des niveaux d'interprétation (voir chap. 1). Dans ce contexte, écrire à un niveau « méta » n'est pas pénalisant.

Enfin, je tiens tout particulièrement à mentionner Gilles Kahn, mon directeur de thèse « de facto », pour son soutien constant, tant sur le plan humain que scientifique. Je crois que peu en auraient fait autant.

— We have not decided yet the existence of God, and you want to eat!
Henry Miller, *Plexus*.

— Cette leçon vaut bien un fromage, sans doute.
La Fontaine, *Fables*.

Introduction

L'objectif principal de cette thèse est de proposer une *formalisation* de l'*évaluation partielle*.

Ce terme d'évaluation partielle regroupe un ensemble de techniques d'optimisation de programmes dont la popularité est allée croissante au cours des quinze dernières années. Si de nombreuses applications concrètes ont d'ores et déjà montré l'intérêt pratique de ces techniques, il reste à notre avis quelques lacunes dans les fondements théoriques, qui entachent certaines mises en œuvre et donnent matière à notre étude.

Notre formalisation pose rigoureusement la problématique de l'évaluation partielle. Elle propose un cadre général pour exprimer la performance d'un programme et pour réaliser des transformations.

Nous fournissons également quelques outils d'aide à la construction d'évaluateurs partiels : correction des transformations, algorithme, terminaison. En revanche, nous ne proposons pas un système « clés en main » qui résoud le problème pour un langage donné. Néanmoins, nous avons une approche algébrique basée en grande partie sur le formalisme des schémas de programme récursifs, que nous pensons suffisamment général pour pouvoir se plier aux spécificités de langages particuliers.

Organisation du document.

Ce document comporte trois parties. La première présente une vue analytique et synthétique des développements actuels de l'évaluation partielle.

1. **Tour d'horizon de l'évaluation partielle.** Après quelques généralités sur les langages de programmation, nous donnons dans le chapitre 1 une première approche de la *problématique* de l'évaluation partielle : il s'agit de déterminer des programmes qui sont à la fois équivalents et plus performants qu'un programme donné. Nous analysons ensuite méthodiquement les courants majeurs : *spécialisation*, *supercompilation*, et *évaluation partielle généralisée*. Le chapitre se conclut sur un bilan comparatif des trois approches. Les déficiences rencontrées motivent le développement des chapitres suivants.

La seconde partie se consacre à la formalisation du problème de l'évaluation partielle.

2. **Sémantique et exécution.** Partant du constat que la performance d'un programme n'est pas intrinsèque mais liée au mode de calcul de ses résultats, nous définissons au chapitre 2 un *modèle d'exécution* afin de spécifier le contenu opératoire d'un programme. Pour être à la fois fidèles à des implémentations réelles, et suffisament abstraits pour pouvoir raisonner, nous fondons notre modélisation sur des sémantiques opérationnelles. Nous proposons notamment l'emploi de la *sémantique naturelle* et du formalisme des *schémas de programmes récursifs*. Un petit langage illustre ces différentes approches.

3. **Mesure de performance.** À l'aide d'un modèle d'exécution, nous indiquons au chapitre 3 comment associer une *mesure de performance* à l'exécution d'un programme. Nous examinons en particulier le cas de la sémantique naturelle et des schémas de programme récursifs, que nous mettons en pratique sur le petit langage défini au chapitre précédent.

   Nous définissons ensuite divers critères pour comparer les performances de deux programmes. Nous examinons minutieusement leurs propriétés et comment ils sont corrélés.

4. **Optimisation et optimalité.** Muni ainsi d'un modèle de performance, nous définissons formellement au chapitre 4 la notion d'*évaluation partielle*. Parce qu'il nous semble raisonnable de chercher des améliorations optimales pour des classes de programmes relativement simples, et de nous contenter de modestes « mieux » pour les classes plus complexes, nous définissons également plusieurs notions d'*évaluation partielle optimale*. Nous illustrons ces définitions à l'aide d'exemples tirés des deux chapitres précédents.

   Nous étudions ensuite des conditions d'existence et de inexistence d'évaluation partielle optimale. Nous définissons un cadre qui permet de transporter des évaluations partielles d'un langage dans un autre, et montrons comment il permet aussi de construire explicitement des programmes optimaux. Enfin, nous disons quel sens donner à l'optimisation d'un ensemble d'exécution et examinons informellement et qualitativement quelques sources d'optimisation.

La troisième partie fournit des outils pour construire un système automatique de transformation de programme.

5. **Transformations et correction.** Nous étudions dans le chapitre 5 la correction des transformations qui combinent pliages et dépliages. Nous puisons dans la littérature plusieurs conditions correctes. L'une d'elles, le *dépliage/pliage faible*, s'applique aux transformations générales de pliage/dépliage pourvu que les règles sémantiques des lois algébriques soient linéaires. Nous donnons quelques propositions sur la commutativité des systèmes de réécriture afin d'affaiblir cette condition en une linéarité à gauche ou à droite.

   Nous définissons ensuite une *séquence de transformations* comme une suite quelconque de transformations élémentaires constituées de pliages, de dépliages, de réécritures selon les lois algébriques, ou de définitions de fonctions auxiliaires. Nous montrons qu'elle peut toujours se ramener à une forme normale qui peut être traitée par les outils présentés précédemment.

6. **Algorithme et terminaison.** Nous étudions enfin l'algorithme de *redémarrage généralisé*, qui permet de piloter une séquence de transformations. Sa terminaison, que nous présentons dans un cadre formel à l'aide d'un bon préordre et d'un ordre bien fondé, n'avait pas jusqu'à lors été rigoureusement démontrée. Enfin, nous analysons quels bons *critères d'arrêt* peuvent équiper un tel algorithme d'évaluation partielle.

Après un bilan de notre approche, nous donnons dans la **Conclusion** quelques perspectives et commentons les limites de l'évaluation partielle. Suivent quelques annexes.

D. Démonstrations auxiliaires. Nous avons rassemblé dans l'annexe D la démonstration de la plupart des propositions élémentaires ; fastidieuses, elles n'apportent pas de compréhension supplémentaire des phénomènes et encombrent inutilement le texte. Nous y avons aussi placé certaines démonstrations « techniques », ainsi que les justifications de quelques affirmations mineures, évoquées dans le texte hors du cadre d'une proposition formelle.

N. Notations, définitions et propositions usuelles. Nous nous sommes efforcés d'employer dans chaque domaine la terminologie et les notations les plus répandues. Avec cela comme excuse, n'avons pas rappelé les définitions courantes dans le texte-même ; celles-ci sont regroupées dans l'annexe N.

Plan de lecture.

Afin de guider une lecture partielle ou discontinue, nous recensons les dépendances entre les différents chapitres ou parties.

Panorama (chap. 1). La lecture du chapitre 1 n'est pas indispensable pour aborder ceux qui suivent ; elle leur sert néanmoins de motivations. Un lecteur étranger au domaine pourra consulter tout d'abord les exemples des sections §1.3.2, §1.4.1 et §1.5.2 afin d'avoir une première idée des tenants et aboutissants.

Formalisation (chap. 2, 3, 4). Les chapitres 2, 3 et 4 forment une suite logique qui permet de construire progressivement une formalisation de l'évaluation partielle ; ils doivent être lus l'un après l'autre. Les seuls renvois au chapitre 1 le sont vers des exemples.

Outils (chap. 5, 6). Bien qu'ils s'enchaînent logiquement, les chapitres 5 et 6 sont indépendants entre eux et peuvent être lus séparément.

Conclusion. La conclusion doit être intelligible à un lecteur familier du domaine qui n'aurait pas lu l'ensemble du document. Elle en résume d'ailleurs le contenu avant d'ouvrir quelques perspectives et de discuter les limites de l'évaluation partielle.

Annexes (ann. D, N). L'annexe D peut être omise dans une première lecture, et réservée à un renvoi occasionnel. L'annexe N n'est à consulter que lorsqu'une notation ou une définition est inconnue.

Conventions et notations.

Les notations les plus courantes sont données à l'annexe N. Les autres sont précisées au fil du texte ou en tête de chapitre. À l'heure où nous mettons sous presse, les termes et les notations de l'annexe N n'ont pas tous encore été reportés dans l'index.

Lorsqu'un terme est d'un usage peu fréquent en français, nous indiquons entre parenthèses son équivalent anglais ; nous l'indiquons aussi lorsque les traductions sont éloignées ou pour permettre une meilleure lecture des ouvrages cités.

D'autre part, lorsqu'une démonstration ne suit pas immédiatement une proposition, un lemme ou un théorème, cela signifie qu'elle a été placée dans l'annexe D afin d'alléger le texte. Nous n'indiquons pas explicitement au lecteur qu'il peut s'y reporter.

Certaines définitions sont suivies (éventuellement en annexe) d'une démonstration. C'est soit parce qu'elles sont de nature algorithmique et qu'elles font intervenir un calcul qui doit être justifié, soit parce qu'elles contiennent une proposition, qui a été mise là pour des raisons de comodité (faible importance, seule proposition concernant la définition).

Chapitre 1

Tour d'Horizon de l'Évaluation Partielle

La *spécialisation* est une technique d'optimisation de programmes employée lorsque l'on connait à l'avance une partie des données. Son principe est d'effectuer le maximum de calculs immédiatement réalisables sur ces données connues afin de réduire la tâche finale, lorsque les données manquantes seront fournies. Elle sert de base à *l'évaluation partielle*, qui s'intéresse plus généralement à l'ensemble des *optimisations de haut niveau*.

Nous présentons dans ce chapitre une vue analytique et synthétique développements actuels de l'évaluation partielle. En particulier, nous examinons avec *une même méthodologie* les spécificités et les points communs des principaux axes de l'évaluation partielle : spécialisation, supercompilation, et évaluation partielle généralisée.

Cette étude fait apparaître, dans un bilan final qui compare ces trois approches, plusieurs déficiences auxquelles nous tentons de trouver des remèdes dans les chapitres suivants.

Organisation du chapitre.

- §1.1 Afin de fixer les notations et le vocabulaire de base, nous donnons tout d'abord quelques généralités sur les *langages de programmation* et les *transformations de programmes*. Nous employons pour cela un formalisme général et intuitif que nous conservons dans toute la suite du document.
- §1.2 Nous définissons ensuite, encore de manière informelle, la *problématique* de l'évaluation partielle, et donnons quelques points de repères historiques.
- §1.3 Une fois ce cadre posé, nous présentons la théorie et la pratique de la *spécialisation*, courant majeur de l'évaluation partielle.
- §1.4 Des transformations plus générales que celles employées en spécialisation, mais aussi plus difficiles à mettre en œuvre, conduisent à la *supercompilation*.
- §1.5 Enfin, l'*évaluation partielle généralisée* exploite l'ensemble des informations d'ordre sémantique pour étendre le champ des transformations de simplifications.
- §1.6 Ce panorama se conclut sur un *bilan comparatif* des différentes approches, qui motive les développements des chapitres suivants.

Notre objectif, dans ces chapitres, est d'offrir à la fois une *formalisation* et des *outils* pour l'évaluation partielle. Nous ouvrons le chapitre 2 sur quelques bases en matière de programmation : sémantique et exécution.

1.1 Langages et transformations.

La présentation qui suit est écrite dans un style *fonctionnel* : un programme représente une fonction qui prend en argument une donnée et retourne un résultat. Néanmoins, elle est transposable en termes *impératifs*, grâce à des fonctions qui opèrent sur une mémoire ou un environnement, ou bien en termes *relationnels* avec des fonctions qui retournent des booléens ou des ensembles de solutions. Au lieu de *fonction*, il faut alors entendre *procédure* ou *prédicat*. Nous ne rappelons pas les notions liées à la calculabilité et à la complexité [HU79], que nous ne faisons qu'effleurer.

1.1.1 Langage de programmation.

Un *langage de programmation* L comprend un ensemble $\mathcal{P}$ de *programmes*, un ensemble $\mathcal{D}$ de *données* et un ensemble $\mathcal{R}$ de *résultats*. Il est caractérisé par sa *sémantique*, une fonction $L : \mathcal{P} \times \mathcal{D} \rightarrow \mathcal{R}$, qui établit un lien entre un programme p, une donnée d et un résultat r : $L(p, d) = r$. Il revient au même de considérer la fonction $\hat{L} : \mathcal{P} \rightarrow (\mathcal{D} \rightarrow \mathcal{R})$; un programme p de $\mathcal{P}$ répresente alors la fonction mathématique $\hat{L}(p) : \mathcal{D} \rightarrow \mathcal{R}$. Ces deux représentations isomorphes (curryfication) sont reliées par $(\hat{L}(p))(d) = L(p, d)$. Par abus de langage, nous omettons souvent l'accent circonflexe et, suivant que nous désirons accentuer l'une ou l'autre formulation, nous notons simplement $L(p, d) = r$ ou

$$L\ p\ d = r$$

La fonction L n'est pas nécessairement totale ; elle est définie sur un *domaine de définition* $Dom(L) \subset \mathcal{P} \times \mathcal{D}$.

$$Dom(L) = \{(p, d) \in \mathcal{P} \times \mathcal{D} \mid \exists r \in \mathcal{R} \quad L\ p\ d = r\}$$

Il en va de même de la *fonction* $L(p)$ attachée à un *programme* p.

$$Dom(p) = Dom(L(p)) = \{d \in \mathcal{D} \mid \exists r \in \mathcal{R} \quad L\ p\ d = r\}$$

Les raisons pratiques qui font que $L\ p\ d$ n'est pas défini sont la non-terminaison et les erreurs de typage. À la différence d'un langage de spécification, la sémantique L d'un *langage de programmation* est une fonction *calculable*. Une sémantique *opératoire* indique *explicitement* comment un programme p permet d'obtenir un résultat r à partir d'une donnée d. Elle s'oppose en cela à une sémantique *déclarative* qui spécifie r sans donner nécessairement un moyen de le calculer. L'*évaluation*, ou l'*exécution*, désigne le processus de calcul de r selon une sémantique opératoire.

1.1.2 Évaluateur.

Lorsque $\mathcal{D}$ comporte des produits cartésiens, nous notons (d_1, d_2) une donnée composée de deux parties d_1 et d_2. L'évaluation s'écrit alors $L\ p\ (d_1, d_2) = r$.

Un *interprète* pour le *langage source* $L_s : \mathcal{P}_s \times \mathcal{D}_s \rightarrow \mathcal{R}_s$, écrit dans un langage $L : \mathcal{P} \times \mathcal{D} \rightarrow \mathcal{R}$, est un programme $interp$ de $\mathcal{P}$ qui simule la sémantique de L_s. Pour tout programme p_s de $\mathcal{P}_s$, et toute donnée d_s de $\mathcal{D}_s$, il vérifie :

$$L\ interp\ (p_s, d_s) = L_s\ p_s\ d_s$$

En réalité, un programme $p_s \in \mathcal{P}_s$ et une donnée $d_s \in \mathcal{D}_s$ ne sont pas nécessairement homogènes à des données de $\mathcal{D}$, pas plus que le résultat $r_s = L_s\ p_s\ d_s \in \mathcal{R}_s$ n'est homogène à un résultat de $\mathcal{R}$. Il faut donc donner des *fonctions de codage*, qui permettent d'interpréter des objets de L_s comme des objets de L, et réciproquement. Ce sont des applications calculables $T_{\mathcal{P}_s} : \mathcal{P}_s \rightarrow \mathcal{D}_1$, $T_{\mathcal{D}_s} : \mathcal{D}_s \rightarrow \mathcal{D}_2$ et $T_{\mathcal{R}} : \mathcal{R} \rightarrow \mathcal{R}_s$. Pour $L_s = $ LISP et $L = $ SML, on peut distinguer par exemple un programme $p_s =$ `(lambda(x) x)` d'une donnée $d = T_{\mathcal{P}_s}(p_s) =$ *LIST[ATOM "lambda", LIST[ATOM "x"], ATOM "x"]* construite à l'aide de constructeurs de types de données. L'interprète $interp$ vérifie alors la relation $T_{\mathcal{R}}(L\ interp\ (T_{\mathcal{P}_s}(p_s), T_{\mathcal{D}_s}(d_s))) = L_s\ p_s\ d_s$. Elle correspond au diagramme suivant.

$$\begin{tikzcd}
& \mathcal{P}_s \arrow[d, "T_{\mathcal{P}_s}"'] & \times & \mathcal{D}_s \arrow[d, "T_{\mathcal{D}_s}"'] \arrow[r, "L_s"] & \mathcal{R}_s \\
\{interp\} \times & (\mathcal{D}_1 & \times & \mathcal{D}_2) \arrow[r, "L"] & \mathcal{R} \arrow[u, "T_{\mathcal{R}}"']
\end{tikzcd}$$

Pour plus de lisibilité, nous omettons le plus souvent la mention des fonctions de codage.

Un *compilateur* est un programme $comp$, écrit dans un langage L, qui traduit des programmes p_s écrits dans un langage source $L_s : \mathcal{P}_s \times \mathcal{D}_s \rightarrow \mathcal{R}_s$ vers des programmes p_c écrits dans un *langage cible* $L_c : \mathcal{P}_c \times \mathcal{D}_c \rightarrow \mathcal{R}_c$. Aux fonctions de codage près, $comp$ vérifie pour toute donnée d_s de $\mathcal{D}_s$:

$$p_c = L\ comp\ p_s$$

$$L_c\ (L\ comp\ p_s)\ d_s = L_s\ p_s\ d_s$$

Si l'on rétablit les fonctions de codage, la compilation correspond au diagramme suivant.

$$\begin{tikzcd}
& & \mathcal{P}_s \arrow[d, "T_{\mathcal{P}_s}"'] & \times & \mathcal{D}_s \arrow[rrr, "L_s"] \arrow[ddrr, "T_{\mathcal{D}_s}"] & & & \mathcal{R}_s \\
\{comp\} & \times & \mathcal{D} \arrow[rr, "L"] & & \mathcal{R} \arrow[d, "T_{\mathcal{R}}"'] & & & \\
& & & & \mathcal{P}_c & \times & \mathcal{D}_c \arrow[r, "L_c"] & \mathcal{R}_c \arrow[uu, "T_{\mathcal{R}_c}"']
\end{tikzcd}$$

Un *évaluateur* pour le langage L_s est un interprète pour L_s ou un compilateur de L_s dans un langage L_c. Il permet de calculer effectivement la sémantique de L_s.

1.1.3 Formalisme sémantique.

L'évaluation partielle est un procédé constructif qui étend la notion d'évaluation. À ce titre, elle demande une sémantique qui ne soit pas purement déclarative (cf. §1.1.1). Cette sémantique peut être elle-même décrite dans le cadre d'un *formalisme sémantique* $F : \mathcal{S} \times (\mathcal{P} \times \mathcal{D}) \rightarrow \mathcal{R}$ où $\mathcal{S}$ désigne un ensemble de *spécification* de langages de programmation.

Une *spécification sémantique* pour le langage $L : \mathcal{P} \times \mathcal{D} \rightarrow \mathcal{R}$, écrite dans le formalisme F, est une spécification s de $\mathcal{S}$ qui exprime la signification de L. Pour tout programme p de $\mathcal{P}$ et toute donnée d de $\mathcal{D}$,

$$F\ s\ (p, d) = L\ p\ d$$

Autrement dit, $L = F(s)$. D'autre part, si l'on peut exprimer le formalisme sémantique F comme un langage de programmation, on reconnaît alors dans la spécification sémantique s un *interprète du langage L, écrit en F* (cf. §1.1.2).

1.1.4 Équivalence.

Deux programmes p et q sont *strictement équivalents* si l'on ne peut les distinguer par L, c'est-à-dire si, appliqués aux mêmes données, ils produisent un même résultat.

$$p \equiv q \quad \text{ssi} \quad Dom(p) = Dom(q) \quad \text{et} \quad L\ p\ d = L\ q\ d \quad \text{pour tout } d \in Dom(p)$$

Cette équivalence stricte signifie que les deux résultats $L\ p\ d$ et $L\ q\ d$ sont simultanément ou bien indéfinis, ou bien définis et égaux entre eux, c'est-à-dire que les fonctions $L(p)$ et $L(q)$ de $\mathcal{D} \rightarrow \mathcal{R}$ sont égales. Plus généralement, la comparaison $\sqsubseteq$ du *comportement opérationnel* global des programmes (ou *extension paresseuse de l'équivalence*) s'exprime ainsi :

$$p \sqsubseteq q \quad \text{ssi} \quad Dom(p) \subset Dom(q) \quad \text{et} \quad L\ p\ d = L\ q\ d \quad \text{pour tout } d \in Dom(p)$$

La relation $p \sqsubseteq q$ signifie $L(p) = L(q)_{|Dom(p)}$, c'est-à-dire que $L\ p\ d$ est indéfini ($L\ q\ d$ peut être défini ou non), ou bien qu'il est défini et égal à $L\ q\ d$ (en ce cas nécessairement défini). De sorte que l'équivalence stricte entre programmes s'écrit aussi :

$$p \equiv q \quad \text{ssi} \quad p \sqsubseteq q \quad \text{et} \quad p \sqsupseteq q$$

Pour comparer deux programmes p et p' écrits dans deux langages différents $L : \mathcal{P} \times \mathcal{D} \rightarrow \mathcal{R}$ et $L' : \mathcal{P}' \times \mathcal{D}' \rightarrow \mathcal{R}'$, il faut donner une correspondance entre les données de $\mathcal{D}$ et $\mathcal{D}'$, et entre les résultat de $\mathcal{R}$ et $\mathcal{R}'$. Il s'agit de fonctions de codage $T_{\mathcal{D}} : \mathcal{D} \rightarrow \mathcal{D}'$ et $T_{\mathcal{R}'} : \mathcal{R}' \rightarrow \mathcal{R}$ (cf. §1.1.2). Une fois ces fonctions fixées, la comparaison du comportement opérationnel global des programmes p et p' s'exprime ainsi :

$$p \sqsubseteq p' \quad \text{ssi} \quad T_{\mathcal{D}}(Dom(p)) \subset Dom(p') \quad \text{et} \quad L\ p\ d = T_{\mathcal{R}'}(L'\ p'\ T_{\mathcal{D}}(d)) \quad \text{pour tout } d \in Dom(p)$$

Cette relation signifie que p' *code* p partout où p est défini.

1.1.5 Transformations de programmes.

Une *transformation de programmes* est une application $T : \mathcal{P} \rightarrow \mathcal{P}'$ qui envoie les programmes p d'un langage L sur les programmes $T(p)$ d'un langage L'. En pratique, les programmes p et $T(p)$ ont comportement opérationnel similaire : une transformation est *valide* (en anglais « sound ») si elle transforme tout programme p en un programmme $T(p)$ de caractérisation moins fine : $T(p) \sqsubseteq p$; en d'autres termes, elle n'introduit pas d'incohérence. Elle est *complète* lorsque $T(p)$ contient toute la caractérisation de p : $p \sqsubseteq T(p)$. Enfin, une transformation est *correcte* lorsqu'elle est valide et complète : $T(p) \equiv p$; on ne peut distinguer les deux programmes par l'observation $\sqsubseteq$.

L'étude d'une transformation de programmes $T : \mathcal{P} \rightarrow \mathcal{P}'$ comporte les étapes suivantes :

Langage et sémantique : L'observation opérationnelle $\sqsubseteq$ est précisée par la donnée des *sémantiques* des langages L et L'.

Transformations : Une *transformation élémentaire locale* $p' = T(p)$ substitue une partie du texte de p par un texte qui lui est équivalent ; il est nécessaire pour cela que $L = L'$. A l'opposé, une *transformation élémentaire globale* nécessite un parcours itératif de p (par exemple pour le calcul d'un point fixe) ; dans ce cas, les langages L et L' ne sont pas nécessairement égaux. Si $L \neq L'$, la transformation $T : \mathcal{P} \rightarrow \mathcal{P}'$ est une *traduction* des programmes de $\mathcal{P}$ en programmes de $\mathcal{P}'$.

Correction : Une transformation correcte fabrique des programmes équivalents : $p \equiv p'$. Dans des cas reconnus, on admet — et même souhaite parfois — des programmes davantage définis : $p \sqsubseteq p'$. On évite par contre les situations où $p' \sqsubseteq p$ car dans ce cas p' est défini moins souvent que p ; il a en quelque sorte perdu de l'information.

Lorsque la transformation est automatisée, il faut également spécifier les points suivants.

Algorithme : Un *algorithme* spécifie une *transformation composée*, suite de transformations choisies dans une gamme de transformations élémentaires. Sans propriété de *confluence*, si plusieurs transformations élémentaires sont *applicables* à diverses étapes, on explicite ou non un choix déterministe parmi ces diverses transformations.

Terminaison : On ne dispose réellement d'un algorithme que lorsque l'on peut garantir la *terminaison* de cette suite de transformations. On atteint un programme *final*, en *forme normale*, lorsqu'aucune des transformations élémentaires n'est applicable au programme courant.

Enfin, il faut justifier l'intérêt de la transformation.

Objet des transformations : L'optimisation a pour but de rendre les programmes plus efficaces. Il faut donc expliciter un *critère de performance* et montrer que les transformations effectivement améliorent les programmes au sens de ce critère.

Conseils et heuristiques : On préconise parfois un style de programmation ou des heuristiques afin d'obtenir de meilleurs résultats. Des paramètres permettent d'ajuster les conditions de terminaison.

C'est la liste des points que nous considèrerons un à un pour analyser les différentes approches de l'évaluation partielle.

1.2 Problématique de l'évaluation partielle.

Dans sa formulation la plus générale, le *problème de l'évaluation partielle* pour un langage $L : \mathcal{P} \times \mathcal{D} \rightarrow \mathcal{R}$ est, pour tout $p \in \mathcal{P}$, de savoir trouver des $p' \in \mathcal{P}$ équivalents à p, qui vérifient un certain critère de performance. Ce critère est généralement lié à l'espace mémoire consommé et/ou au temps d'exécution ; en de rares occasions, ce critère exprime que p' est sous une forme canonique. Un *évaluateur partiel* est un programme qui calcule *une* solution d'un problème d'évaluation partielle.

L'évaluation partielle partage certains des objectifs de la *compilation*. En effet, celle-ci poursuit un double but de traduction et d'optimisation : elle transforme un programme p de L en un programme équivalent p' de L', choisi pour son efficacité. Dans un certain sens, l'évaluation partielle est un cas particulier de compilation où langage source et langage cible coïncident. À l'inverse, on peut considérer la compilation comme une traduction « pure », précédée ou suivie d'une phase d'optimisation par évaluation partielle. Il est donc naturel de voir les deux domaines s'échanger vocabulaire et techniques.

Il est plaisant de faire remonter l'*évaluation partielle*[1] au théorème de curryfication récursive (dit théorème S_n^m) de Kleene [Kle52]. En pratique, ce n'est qu'au début des années 60 que Lombardi et Raphael réalisent le premier évaluateur partiel, basé sur LISP. Les années 70 sont marquées par les projections de Futamura et les premières applications importantes à l'optimisation. Les premiers évaluateurs partiels auto-applicables et l'analyse de temps de liaison doivent attendre les années 80 ; c'est le triomphe de la spécialisation. Nouvelles techniques et applications foisonnent en ce début des années 90, sans qu'émergent véritablement d'idée phare ; nous sommes dans une phase de maturation.

La bibliographie annotée [SZ88] donne les références essentielles de cette rapide chronologie. Consel et Danvy donnent dans [CD93] un aperçu des avancées récentes, avec un accent particulier sur la spécialisation. Nous citons quelques autres références dans les sections suivantes, qui sont consacrées respectivement à la spécialisation, à la supercompilation, et à l'évaluation partielle généralisée.

1.3 Spécialisation.

L'évaluation partielle doit beaucoup à l'« école danoise », menée par Neil Jones, qui a non seulement défriché et mis en œuvre de nombreux aspects de la *spécialisation*, mais a aussi eu de multiples actions de popularisation [BEJ88, EBF88, PEP93, JGS93]. Une partie de la présentation qui suit est d'ailleurs inspirée de certains de ses travaux [JSS85, Mog89a].

La section §1.3.1 suivante, qui présente les grands principes de la spécialisation, est relativement aride pour un lecteur qui n'est pas un habitué du domaine. Il aura intérêt à se référer aux exemples et aux explications des sections §1.3.2 et §1.3.3.

1.3.1 Principes de la spécialisation.

La spécialisation repose sur l'isomorphisme entre $\mathcal{P} \times \mathcal{D} \to \mathcal{R}$ et $\mathcal{P} \to (\mathcal{D} \to \mathcal{R})$, et sur l'identification entre programmes et données. Un *programme résiduel* d'un programme p de L, par rapport à la donnée partielle d_1, est un programme p_{d_1} de L tel que pour toute donnée d_2 complétant d_1 en $d = (d_1, d_2)$, on ait :

$$L\ p_{d_1}\ d_2 = L\ p\ (d_1, d_2)$$

On dit que p_{d_1} est *spécialisé* par rapport à la donnée *statique* d_1 ; la donnée d_2 est qualifiée de *dynamique*. Bien que le domaine de p_{d_1} soit différent de celui de p, il est conceptuellement équivalent à p sur les données d qui comportent la donnée partielle d_1. Le programme résiduel n'est pas unique. Il existe un *programme résiduel trivial* q défini par : $q\ d = p\ (d_1, d)$; il « attend » en quelque sorte qu'on lui fournisse la donnée d_2 pour exécuter le programme d'origine p sur les données (d_1, d_2). Plus généralement, dans le cas où le programme p prend n arguments, les données $d_1, \ldots, d_n$ sont scindées en deux ensembles disjoints, données statiques et données dynamiques.

Un *spécialiseur* est un programme mix[2] écrit dans un langage L' qui produit des programmes résiduels p_{d_1} à partir de programmes p et de données d_1 écrits dans un langage L. Autrement dit, pour tout programme p et

[1] Le terme d'*évaluation* a pour origine les premiers langages abordés, qui s'appuyaient sur une sémantique purement procédurale. En programmation logique, on distingue à présent l'*évaluation partielle* pour PROLOG-*procédural* et la *déduction partielle* pour PROLOG-*déclaratif*. La *spécialisation*, à l'origine de la discipline, a laissé le qualificatif *partielle*, car elle étudie les programmes dont les données sont *partiellement* connues.

[2] Cette dénomination provient de la notion de *mixed computation* [Ers78].

toutes données (d_1, d_2), mix vérifie les équations :

$$L'\ mix\ (p, d_1) = p_{d_1}$$

$$L\ (L'\ mix\ (p, d_1))\ d_2 = L\ p\ (d_1, d_2)$$

Le *spécialiseur trivial* est celui qui « calcule » le programme résiduel trivial. Si l'on rétablit les fonctions de codage (cf. §1.1.2), cette équation correspond au diagrame suivant (*Id* désigne l'application identité).

$$\begin{tikzcd}
\mathcal{P} \arrow[d, "T_{\mathcal{P}}"'] & \times\ (& \mathcal{D}_1 \arrow[d, "T_{\mathcal{D}_1}"'] & \times & \mathcal{D}_2\ ) \arrow[rrrr, "L"] \arrow[ddrrr, "Id"] & & & & \mathcal{R} \\
\{mix\} & \times\ (& \mathcal{D}'_1 & \times & \mathcal{D}'_2\ ) \arrow[r, "L'"] & \mathcal{R}' \arrow[d, "T_{\mathcal{R}'}"'] & & & \\
& & & & & \mathcal{P} & \times & \mathcal{D}_2 \arrow[r, "L"] & \mathcal{R} \arrow[uu, "Id"']
\end{tikzcd}$$

Soit $interp$ un interprète, écrit en L, d'un langage source $L_s : \mathcal{P}_s \times \mathcal{D}_s \to \mathcal{R}_s$. Par définition, il vérifie $L\ interp\ (p_s, d_s) = L_s\ p_s\ d_s$. Appliquons la définition de mix à $p = interp$ et $d_1 = p_s \in \mathcal{P}_s$. On peut calculer $interp_{p_s} = L'\ mix\ (interp, p_s)$. Ce programme $interp_{p_s}$ vérifie pour toute donnée d_s de $\mathcal{D}_s$ l'équation $L\ interp_{p_s}\ d_s = L_s\ p_s\ d_s$. On y reconnait le résultat de la compilation de L_s vers le langage cible $L_c = L$:

$$p_c = L'\ mix\ (interp, p_s)$$

Cette équation porte le nom de *première projection de Futamura* [Fut71]. Elle indique comment mix permet de compiler de L_s vers L si l'on connait un interprète $interp$ de L_s écrit en L.

$$\begin{tikzcd}
& & \mathcal{P}_s \arrow[d, "T_{\mathcal{P}_s}"'] & \times & \mathcal{D}_s \arrow[d, "T_{\mathcal{D}_s}"] \arrow[rrrr, "L_s"] & & & & \mathcal{R}_s \\
\{interp\} \arrow[d, "T_{\mathcal{P}}"'] & \times\ (& \mathcal{D}_1 \arrow[d, "T_{\mathcal{D}_1}"'] & \times & \mathcal{D}_2\ ) \arrow[rrrr, "L"] \arrow[ddrrr, "Id"] & & & & \mathcal{R} \arrow[u, "T_{\mathcal{R}}"'] \\
\{mix\} & \times\ (& \{interp'\} & \times & \mathcal{D}'_2\ ) \arrow[r, "L'"] & \mathcal{R}' \arrow[d, "T_{\mathcal{R}'}"'] & & & \\
& & & & & \mathcal{P} & \times & \mathcal{D}_2 \arrow[r, "L"] & \mathcal{R} \arrow[uu, "Id"']
\end{tikzcd}$$

On peut également remarquer que $p_c = L'\ mix\ (interp, p_s) = L'\ mix_{interp}\ p_s$.

Supposons à présent que les langages L et L' sont égaux. Nous définissons un programme $comp$ de $L = L'$ de la manière suivante.

$$comp = mix_{interp} = L\ mix\ (mix, interp)$$

Cette nouvelle équation est la *deuxième projection de Futamura*. Elle décrit comment mix permet de fabriquer un compilateur $comp$ de L_s vers L si l'on connait un interprète $interp$ de L_s écrit en L, car $p_c = L\ comp\ p_s$.

$$
\begin{tikzcd}
 & & \mathcal{P}_s \times \arrow[d, "\mathcal{T}_{\mathcal{P}_s}"'] & \mathcal{D}_s \arrow[rr, "L_s"] \arrow[d, "\mathcal{T}_{\mathcal{D}_s}"] & & \mathcal{R}_s \\
 & \{interp\} \times \arrow[dl, "\mathcal{T}_{\mathcal{P}}"'] & (\mathcal{D}_1 \times \arrow[dd, "Id"] & \mathcal{D}_2) \arrow[rr, "L"] \arrow[dddr, "Id"] & & \mathcal{R} \arrow[u, "\mathcal{T}_{\mathcal{R}}"'] \\
\{mix\} \times (\{mix\} \times \{interp'\}) \arrow[r, "L"] & \{comp\} \arrow[d, "\mathcal{T}_{\mathcal{R}'}"'] & & & & \\
 & \{comp'\} \times & \mathcal{D}_1 \arrow[r, "L"] & \mathcal{P} \arrow[d, "\mathcal{T}_{\mathcal{R}'}"'] & & \\
 & & & \mathcal{P} \times & \mathcal{D}_2 \arrow[r, "L"] & \mathcal{R} \arrow[uuu, "Id"']
\end{tikzcd}
$$

Notons enfin que $comp = L\ mix\ (mix, interp) = L\ mix_{mix}\ interp$ et posons :

$$cogen = mix_{mix} = L\ mix\ (mix, mix)$$

Cette dernière équation s'appelle la *troisième projection de Futamura*. Elle exprime comment mix permet de construire un générateur de compilateurs $cogen$ de L_s vers L si l'on connait un interprète $interp$ de L_s écrit en L, car $comp = L\ cogen\ interp$.

$$
\begin{tikzcd}
 & & & \mathcal{P}_s \times \arrow[d, "\mathcal{T}_{\mathcal{P}_s}"'] & \mathcal{D}_s \arrow[rr, "L_s"] \arrow[d, "\mathcal{T}_{\mathcal{D}_s}"] & & \mathcal{R}_s \\
\{mix\} \times (\{mix\} \times \{mix\}) \arrow[r, "L"] & \{cogen\} \arrow[dl, "\mathcal{T}_{\mathcal{R}'}"'] & \{interp\} \times \arrow[dl, "\mathcal{T}_{\mathcal{P}}"'] & (\mathcal{D}_1 \times \arrow[dd, "Id"] & \mathcal{D}_2) \arrow[rr, "L"] \arrow[dddr, "Id"] & & \mathcal{R} \arrow[u, "\mathcal{T}_{\mathcal{R}}"'] \\
\{cogen'\} \times & \{interp'\} \arrow[r, "L"] & \{comp\} \arrow[d, "\mathcal{T}_{\mathcal{R}'}"'] & & & & \\
 & & \{comp'\} \times & \mathcal{D}_1 \arrow[r, "L"] & \mathcal{P} \arrow[d, "\mathcal{T}_{\mathcal{R}'}"'] & & \\
 & & & & \mathcal{P} \times & \mathcal{D}_2 \arrow[r, "L"] & \mathcal{R} \arrow[uuu, "Id"']
\end{tikzcd}
$$

La possibilité d'appliquer mix à lui-même est appelée *auto-application*. Dû aux techniques employées, il peut être avantageux parfois d'écrire $cogen$ directement, plutôt que mix, afin d'éliminer des redondances qui détériorent la performance de l'évaluateur partiel [BW93] (mais pas de l'évaluation partielle calculée).

Il faut noter qu'un spécialiseur mix de L écrit en L et un interprète $interp$ de L_s également écrit en L ne suffisent pas *seuls* à fabriquer leurs équivalents dans L_s. Aucune équation ne fournit mix_s (spécialiseur pour le langage L_s écrit en L_s), ni $interp_s$ (interprète de L_s écrit en L_s).

1.3.2 Exemples de spécialisation.

Voici quelques exemples élémentaires de spécialisation pour le langage STANDARD ML[3]. Nous conservons dans cette section le style proche de LISP (avec des primitives, sans motifs ni définitions clausales) employé dans [Mog89a], dont ces exemples sont tirés. Nous examinons diverses spécialisations de la fonction `append` suivante, qui réalise la concaténation des listes :

```
fun append(x,y) = if (null x) then y else (hd x)::append(tl x,y)
```

Nous supposons donc connu l'un ou l'autre des deux arguments de `append`, et nous examinons quels types d'optimisation on peut réaliser quand on dispose de cette information.

Spécialisation triviale. La spécialisation la plus simple consiste à ne rien faire, c'est-à-dire à attendre l'arrivée des données manquantes. Elle est analogue aux *fermetures* utilisées pour la curryfication dans les langages fonctionnels. La fonction `append_y12` suivante est la *spécialisation triviale* (cf. §1.3.1) de `append` par rapport à un second argument `y` égal à la liste `[1,2]`.

```
fun append_y12(x) = append(x,[1,2])
```

Propagation des constantes. Si une fonction définie par $f(x_1, \ldots, x_n) = exp$ est toujours appelée avec un même argument $x_i = c$, on peut remplacer chaque apparition de x_i dans exp par c, et ôter x_i de la liste $(x_1, \ldots, x_n)$ des paramètres de f. En pratique, on suit les étapes suivantes (les parties importantes qui interviennent dans la transformation sont soulignées).

```
0)  fun append_y12(x) = append(x,[1,2])
1)  fun append_y12(x) =
      if (null x) then [1,2] else (hd x)::append(tl x,[1,2])
2)  fun append_y12(x) =
      if (null x) then [1,2] else (hd x)::append_y12(tl x)
```

L'étape (1) est un *dépliage* de la fonction `append` : on a remplacé l'appel de `append` dans l'expression `append(x,[1,2])` par le corps de sa définition où la variable `y` a été remplacée par l'expression `[1,2]`. Plus généralement, si une fonction est définie par $f(x_1, \ldots, x_n) = exp$, le *dépliage de* f consiste à remplacer une expression $f(exp_1, \ldots, exp_n)$ par $exp[x_1/exp_1, \ldots, x_n/exp_n]$, terme qui désigne l'expression exp où les variables $x_1, \ldots, x_n$ ont été remplacées respectivement par $exp_1, \ldots, exp_n$.

L'étape (2) correspond à un *pliage* de `append_y12` ; c'est l'opération inverse du dépliage, qui consiste à reconnaître le corps instancié de la définition d'une fonction et à le remplacer par l'appel fonctionnel qui lui correspond. Autrement dit, si l'on dispose d'une fonction définie par $f(x_1, \ldots, x_n) = exp$, le *pliage de* f dans une expression qui s'écrit $exp[x_1/exp_1, \ldots, x_n/exp_n]$ consiste à la remplacer par l'expression $f(exp_1, \ldots, exp_n)$.

Le gain de cette opération est ici simplement la réduction du nombre d'arguments et du nombre d'accès à certaines variables. Plus généralement, le but de la propagation des constantes est de faire apparaître des

[3] STANDARD ML, abrégé en SML, est un langage fonctionnel fortement typé. Les fonctions `nil`, `::` (notée de manière infixe), `hd`, `tl`, `null` et correspondent respectivement aux fonctions `nil`, `cons`, `car`, `cdr` et `null` de LISP. L'abstraction `(lambda (x)` exp`)` s'écrit `fn x =>` exp . Pour plus de détails, on peut consulter par exemple [Pau91, MTH90].

expressions *statiques*, c'est-à-dire des expressions qui ne comportent pas de variables, qui sont donc indépendantes de la valeur des paramètres d'appel et qui par conséquent peuvent être immédiatement évaluées. C'est le cas dans l'exemple suivant. En revanche, pour évaluer une expression *dynamique*, il faut connaître la valeur des paramètres d'appel.

Propagation des constantes avec dépliage. Spécialisons à présent `append` par rapport à un *premier* argument `x` = `[1,2]`. On peut dérouler certains calculs sur les objets statiques :

```
0)  fun append_x12(y) = append([1,2],y)
1)  fun append_x12(y) =
      if (null [1,2]) then y else (hd [1,2])::append(tl [1,2],y)
2)  fun append_x12(y) = 1::append([2],y)
3)  fun append_x12(y) =
      1::(if (null [2]) then y else (hd [2])::append(tl [2],y))
4)  fun append_x12(y) = 1::2::append([],y)
5)  fun append_x12(y) =
      1::2::(if (null []) then y else (hd [])::append(tl [],y))
6)  fun append_x12(y) = 1::2::y
```

Le dépliage de la définition de `append` en (1) fait apparaître des expressions statiques qui sont ensuite évaluées en (2). On réitère deux fois le même procédé pour atteindre finalement la spécialisation (6). La fonction `append_x12` résultante est plus efficace parce qu'un certain nombre de tests et d'appels de primitives ont été réalisées à l'avance ; il n'y a plus d'appels fonctionnels dérivés de `append`.

Spécialisation polyvariante. Une spécialisation *monovariante* ne permet de fabriquer qu'*une seule* version spécialisée de la fonction originale. En revanche, une spécialisation *polyvariante* [Bul84] autorise plusieurs instances : un spécialiseur peut construire au cours de la même session plusieurs spécialisations $f_{d_1}, \ldots, f_{d_n}$ d'une même fonction f. Soit par exemple `ack` la fonction de Ackermann :

```
fun ack(x,y) = if x=0 then y+1
          else if y=0 then ack(x-1,1)
          else             ack(x-1,ack(x,y-1))
```

Sa spécialisation polyvariante `ack2` par rapport à `x` = `2` conduit à

```
fun ack0 y = y+1
fun ack1 y = if y=0 then 2 else ack0(ack1(y-1))
fun ack2 y = if y=0 then 3 else ack1(ack2(y-1))
```

L'appel de `ack2(y)` est plus efficace que celui de `ack(2,y)` pour des raisons identiques aux cas précédents. Une spécialisation monovariante n'aurait produit que :

```
fun ack2 y = if y=0 then 3 else ack(1,(ack2(y-1)))
```

où les appels `ack(0,y)` et `ack(1,y)` n'ont pas été spécialisés. Toutefois, la distinction entre spécialisation monovariante et spécialisation polyvariante est purement technique et due principalement à l'*analyse de temps de liaison* (cf. §1.3.4, algorithme), dont le modèle d'origine a dû être étendu afin de permettre plusieurs variantes spécialisées d'une même fonction.

1.3.3 Applications de la spécialisation.

L'essence de la spécialisation est de supprimer les *éléments interprétés* d'un programme. Ses applications sont contenues dans les trois projections de Futamura. On peut en trouver des exemples variés dans [BEJ88, EBF88, PEP91, PEP92, PEP93]. Certaines sont à la frontière d'un emploi industriel. Voici les plus communes.

L'analyse lexicale est un des domaines de prédilection de la spécialisation, dont on ne sait s'il est un cas d'école spectaculaire ou une preuve tangible de bien-fondé. Soit $\texttt{analex}(reg, str)$ un programme d'un langage L qui décide si une chaîne de caractères str est dans le langage engendré par une expression régulière reg. Pour une expression régulière reg fixée, un programme résiduel $\texttt{analex}_{reg}(str)$ décide de manière équivalente si la chaîne str appartient ou non à reg. Des spécialiseurs mix parviennent à produire un programme résiduel $\texttt{analex}_{reg} = L\ mix\ (\texttt{analex}, reg)$ qui a la forme d'un automate fini, et qui est donc particulièrement efficace. De plus, $L\ mix\ (mix, \texttt{analex}) = mix_{\texttt{analex}}$ est un compilateur d'expressions régulières similiaire à LEX sous UNIX. En effet, $mix_{\texttt{analex}}(reg) = \texttt{analex}_{reg}$. L'analyse syntaxique (parsing) est traitée à l'avenant.

Dans un domaine voisin, on trouve également la compilation de recherche de motif (pattern matching) [Dan91, QG92]. Des spécialiseurs « redécouvrent » l'algorithme de Knuth-Morris-Pratt, de complexité $\mathcal{O}(m+n)$ où m est la taille du motif et n celle du texte, en partant de l'algorithme naïf de complexité $\mathcal{O}(mn)$ [CD89, Smi91].

Dans le domaine des langages de programmation, la spécialisation s'applique, grâce aux projections de Futamura, à la compilation et à la génération de compilateurs à partir d'interprètes [JSS89, CK91b, CD91b]. Elle permet par exemple de compiler l'instrumentation (debugging, profiling, etc.) [Con91] et l'héritage dans les langages orientés-objet [KS91, HM92]. Elle peut aussi déterminer des informations liées à la notion de paresse [HG91, Jør92].

Elle s'applique également à la construction de programmes incrémentaux [Sun91], à l'optimisation de techniques de réécriture [Bon88, SSD91], et à certains calculs numériques [Ber90, BW90, Mog86].

Par ailleurs, compiler, puis exécuter le progamme compilé, prend souvent moins de temps *au total* qu'interpréter le programme original. Un résultat similaire s'observe en évaluation partielle. Connaissant l'ensemble des données d'un programme, calculer le programme résiduel sur une portion « choisie » de ces données, puis l'exécuter sur les données restantes, prend parfois globalement moins de temps qu'exécuter le programme initial sur l'ensemble des données.

1.3.4 Analyse de la spécialisation.

Nous analysons à présent la spécialisation suivant le modèle proposé à la section §1.1.5.

Langage et sémantique.

De nombreux langages ont désormais leur spécialiseurs : des langages fonctionnels d'ordre supérieur comme SCHEME [Bon91, Con90a], des langages de programmation logique comme PROLOG [Has88, Sah91a] ou avec des contraintes [FOF88, HS91], des langages impératifs comme PASCAL [Mey91] ou C [And92], des langages fortement typés comme ML [Lau91b, BW93].

Les spécialiseurs ne travaillent parfois que sur un sous-ensemble de langages réels, encore trop complexes. Du fait de leur relative simplicité, SCHEME et PROLOG sont les plus populaires, notamment en ce qui concerne l'auto-application.

Transformations.

Les transformations de base de la spécialisation sont l'*introduction de nouvelles définitions* (instances d'appels de fonctions déjà existantes), le *dépliage* (en anglais, unfold), l'*évaluation d'expressions statiques* et le *pliage* (fold) des nouvelles définitions. Ce sont toutes des transformations locales. S'y ajoutent parfois des *transformations paresseuses* comme « `hd(`x`::`l`)` $\rightarrow x$ »[4], également locales.

Correction.

Les transformations de dépliage et de pliage ne sont pas correctes (cf. §1.1.5). Cependant, le dépliage est complet et le pliage valide. Soient par exemple les fonctions SML suivantes.

```
fun incr(x)  = x+1
fun loop()   = 1::loop()
fun foo(x,y) = incr(y)
```

Alors que l'évaluation de l'expression `foo(loop(),2)` ne se termine pas (elle *boucle*), le dépliage de `foo` construit l'expression `incr(2)`, qui s'évalue bien en *3*. le dépliage n'est donc pas une transformation valide. De même, si l'on replie la fonction `incr` sur elle-même, on fabrique une fonction « `fun incr(x) = incr(x)` », dont l'évaluation ne se termine jamais. Le pliage n'est donc pas complet.

Néanmoins, si l'on évite l'effacement d'expressions par dépliage, et si l'on se restreint au pliage des nouvelles définitions (après dépliage dans le cas du corps d'une nouvelle définition elle-même), ces transformations sont alors correctes ; elles fabriquent des programmes strictement équivalents aux programmes originaux. Les spécificités de PROLOG traduisent ces condition ainsi : il est toujours correct de plier un prédicat qui n'est défini que par une clause unique, à condition que toutes les variables qui apparaissent dans le corps de cette définition apparaissent également dans la tête et qu'on ne le replie pas sur lui-même. Ces conditions peuvent être affinées [TS84, GS91, PP91a].

Les transformations paresseuses sont par définition uniquement complètes ; elles peuvent produire des programmes de domaine de définition plus large. Par exemple, l'évaluation de l'expression `hd(1+2::loop())` boucle. En revanche, après l'application de la transformation paresseuse « `hd(`x`::`l`)` $\rightarrow x$ », l'expression résultante `1+2` s'évalue en *3*.

Algorithme.

La spécialisation est totalement assujetie aux données connues. Une première stratégie, dite « *online* », consiste à simuler dynamiquement d'évaluation du programme, afin de déterminer au fur et à mesure les transformations à lui appliquer [WCRS91]. Plus coûteuse, mais intrinsèquement plus puissante, elle a eu la réputation arbitraire de produire de gros programmes, lents et peu spécialisés.

[4] Cette transformation est *paresseuse* parce qu'elle élimine l'évaluation de la liste l.

C'est pourquoi beaucoup lui préfèrent encore une stratégie « *offline* », qui précède la spécialisation d'une phase d'*analyse de temps de liaison* (binding time analysis, abrégé en BTA). Connaissant la répartition statique/dynamique des données, celle-ci annote et distingue les parties du programme qui seront connues au moment de la spécialisation proprement dite, et sur lesquelles s'appliqueront les transformations, des parties dynamiques qui resteront inconnues et inchangées. Cette phase d'interprétation abstraite préliminaire a pu conduire à des spécialiseurs auto-applicables efficaces, tant par leur vitesse que par la qualité des programmes spécialisés produits [Bon93, Con90b]. Pour plus de détails, nous renvoyons le lecteur par exemple à [Mog89a].

Terminaison.

Le problème de la terminaison est souvent éludé par les spécialiseurs [Bon93], qui fondent l'arrêt sur la fin du traitement des données statiques. Même lorsque qu'un critère garantit un nombre fini de dépliages — et c'est une propriété rare [WCRS91] — le principe consistant à complètement évaluer toute expression totalement statique compromet la terminaison du processus de spécialisation : un tel spécialiseur peut « boucler ». Si cela se produit dans des parties du code qui ne sont jamais atteintes, il y a conflit radical entre un programme résiduel théorique qui se termine quelles que soient les données dynamiques fournies et un spécialiseur qui boucle sans produire de programme spécialisé.

Ce sont jusqu'ici les spécialiseurs « online » qui offrent le plus souvent de réelles garanties de terminaison car ils conservent davantage d'information que les spécialiseurs « offline », bâtis pour la vitesse. Ils mettent pour cela des bornes à la profondeur de spécialisation, en examinant éventuellement l'historique des dépliages. Par exemple, Sahlin [Sah91b] borne pour PROLOG le nombre d'appel d'un même prédicat sans qu'il y ait décroissance de taille des arguments, la profondeur jusqu'à laquelle les termes sont comparés pour le test de boucle, et l'intervalle de variation sûr des entiers, en dehors duquel on considère qu'il y a risque de boucle. Ce dernier point correspond en fait au cas particulier d'une fonction dont les arguments varient dans un ensemble fini ; lorsque cet ensemble est infini, comme c'est le cas des entiers, on donne une relation d'équivalence d'index fini afin d'obtenir la même propriété : par exemple, la relation « $n \equiv m \;\text{ ssi }\; n = m$ ou $|n|, |m| > 2$ » considère les entiers comme l'ensemble fini $\{-2, -1, 0, 1, 2, \texttt{autres}\}$. Toujours pour PROLOG, Prestwich [Pre92] limite la taille des termes et le nombre d'instances différentes d'une même fonction spécialisée. Il existe des études générales sur le *test de boucle* (loop checking) [BAK91] qui cherchent à établir non seulement les cas où un programme ne boucle pas, mais aussi les cas où il boucle avec certitude. Ce type de test est utilisé pour la sémantique déclarative de PROLOG non seulement pour assurer la terminaison de l'algorithme de transformation, mais aussi pour simplifier les programmes résultants [Bol91].

Lorsqu'un critère d'arrêt est atteint, on stoppe les dépliages et l'on construit, faute de mieux, un appel aux fonctions non spécialisées. Certains spécialiseurs [WCRS91] emploient un mécanisme de *redémarrage généralisé* (cf. §1.4.3), emprunté à la supercompilation. C'est une opération plus courante chez les spécialiseurs PROLOG [Has88, Sah91a].

Objet des transformations.

La spécialisation vise avant tout à réduire le temps d'exécution des programmes. En règle générale, l'évaluation d'expressions statiques élimine des calculs, le dépliage élimine des coûts d'appel de procédure, et le pliage des définitions spécialisées réduit le nombre d'arguments à transmettre. Néanmoins, des transformations de dépliage peuvent dupliquer des calculs. Considérons par exemple [JSS85] la fonction `f` définie par le programme suivant.

```
fun g(n) = n + n
fun f(n) = if n = 0 then 1 else g(f(n-1))
```

Elle ne doit pas être dépliée en :

```
fun f'(n) = if n = 0 then 1 else f'(n-1) + f'(n-1)
```

En effet, la fonction `f` d'origine est linéaire ; après transformation, elle devient exponentielle. La relation entre les deux formes `f` et `f'` est de même nature qu'entre les versions linéaire et exponentielle de la fonction de Fibonaci.

Très peu d'études cependant, aussi bien spécifiques que générales, formalisent ou simplement s'intéressent au gain obtenu par spécialisation.

Si on limite les transformations au pliage, au dépliage et à l'évaluation d'expressions statiques, Andersen et Gomard montrent [AG92] pour un petit langage impératif, mais avec une méthode assez générale, que le gain est au plus linéaire. Par contre, des transformations paresseuses et l'élimination d'expressions répétées, comme par exemple « *exp*`*`*exp* $\rightarrow$ `let val x =` *exp* `in x*x end` » peuvent conduire à des gains supralinéaires. C'est la situation inverse du dépliage de la fonction `g` dans la définition de `f` ci-dessus.

Amtoft donne un résultat similaire à propos de la programmation logique [Amt91], avec une étude complémentaire du cas particulier de plusieurs niveaux d'interprétations. Les hypothèses sont toutefois assez fortes car la quantité mesurée est le nombre d'inférences logiques, ce qui revient à considérer l'unification comme une opération de temps constant, assomption notable.

Dans le cadre du λ-calcul, Nielson compare des programmes résiduels en termes de réductions, au moyen d'un système de types qui représente l'information de temps de liaison [Nie88]. Il reste cependant éloigné de l'aspect pratique des implémentations.

Malmkjær vérifie a posteriori que les « éléments » qui constituent les données statiques ont bien disparu du programme résiduel [Mal92]. Ces éléments sont déterminés automatiquement par une interprétation abstraite du programme et des données statiques. Le résultat de cette analyse est une grammaire qui reflète les conditions syntaxiques que doit vérifier le programme résiduel.

Conseils et heuristiques.

La qualité de la spécialisation est très sensible au style de programmation. Afin d'obtenir de bons résultats, une discipline rigoureuse est parfois préconisée. Ces recommandations se ramènent en pratique à séparer au mieux les parties statiques des parties dynamiques. Dans le cas d'un langage fonctionnel, l'usage d'un « style de passage à la continuation » (continuation passing style) est également conseillé, notamment en ce qui concerne la représentation d'environnements et le retour de résultats multiples [Bon93].

Les spécialiseurs ne garantissent pas une optimisation systématique. Ils obtiennent toutefois des améliorations notables pour un grand nombre d'exemples non-triviaux [Sah91a]. Avec ces systèmes, sont montrés sur des exemples de cas de perte de performance ou des risques de croissance démesurée du code. Des paramètres ajustables bornent des tailles pour la terminaison (lorsqu'elle est garantie) et l'explosion de code.

1.3.5 Étendre la spécialisation.

Les développements de la spécialisation portent en grande partie sur des extensions de l'analyse de temps de liaison et sur quelques nouvelles transformations. En voici quelques unes.

Valeurs partiellement statiques.

Tout ce qui est dynamique est obstacle à la spécialisation ; on ne déroule les calculs que lorsque l'on dispose d'expressions totalement statiques. Or il est fréquent que des données structurées ne soient que partiellement connues. On « déstructure » ces valeurs grâce à une représentation de leur répartition statique/dynamique : grammaires d'arbre [Mog89a], projections [Lau91a], accroissement d'arité [Rom88, Mog89a]. Puisque l'arité des fonctions peut augmenter, le pliage des fonctions spécialisées n'apporte pas un gain systématique.

Valeurs conditionellement statiques.

Certaines valeurs sont conditionnellement statiques. C'est le cas par exemple d'une expression comme `f(if` a `then` b `else` c`)` où b et c sont statiques, et a dynamique. Il est plus avantageux [Mog89a] de la récrire ainsi :

$$\texttt{f(if } a \texttt{ then } b \texttt{ else } c\texttt{)} \quad \rightarrow \quad \texttt{if } a \texttt{ then (f } b\texttt{) else (f } c\texttt{)}$$

On peut désormais évaluer statiquement `(f` b`)` et `(f` c`)`. Cette transformation est en fait un cas particulier du *driving* utilisé en supercompilation (cf. §1.4.3). On peut obtenir un résultat analogue en réécrivant le programme original dans un style de passage à la continuation [CD91a] ; l'expression initiale devient :

$$\texttt{f(if } a \texttt{ then } b \texttt{ else } c\texttt{)} \quad \rightarrow \quad \texttt{fn x => if } a \texttt{ then (f } b \texttt{ x) else (f } c \texttt{ x)}$$

Évaluation partielle paramétrable.

L'*interprétation abstraite* permet de connaître des renseignements d'ordre général sur les valeurs d'un programme, comme par exemple le signe ou l'intervalle de variation d'expressions numériques. L'*évaluation partielle paramétrable « online »* [CK91a] utilise ces informations pour optimiser le programme original en simplifiant les expressions dont les propriétés abstraites suffisent à connaître la valeur. Par exemple, si l'on sait qu'une variable est strictement positive, tester son égalité à zéro retourne toujours la valeur *false*. C'est en fait un cas particulier d'évaluation partielle généralisée (cf. §1.5).

Il existe une version « offline » [CK91a] de l'évaluation partielle paramétrable. Elle étend l'analyse de temps de liaison en une *analyse de facettes*. Connaissant les propriétés abstraites des données connues, les expressions abstraites simplifiables sont marquées statiques. Le spécialiseur se charge ensuite, au moyen de ces annotations, d'effectuer les transformations nécessaires.

1.4 Supercompilation.

Au lieu de spécialiser les fonctions par rapport à des valeurs, V. Turchin propose une spécialisation par rapport à des motifs d'expressions quelconques, des *configurations* ; il lui donne le nom de *supercompilation*

[Tur85, Tur86]. Dans ce modèle, les données statiques ne sont qu'un cas particulier, traité au même titre que le reste du programme.

1.4.1 Exemples de supercompilation.

A titre d'exemple, nous indiquons comment la configuration `append(append(`x`,`y`),`z`)` est transformée par supercompilation en une fonction qui incarne en substance l'appel `append(`x`,append(`y`,`z`))`. La configuration résultante est plus efficace (en temps comme en espace) que l'expression initiale car elle ne copie qu'une fois la liste x au lieu de deux. En pratique, on transforme la fonction `append3` suivante :

```
fun append3(x,y,z) = append(append(x,y),z)
```

Les spécificités du langage dictent différents *styles* de supercompilation, suivant qu'elles privilégient les *sélecteurs de données*, les *définitions clausales* ou les *primitives*.

Style « sélecteur de données ».

Les motifs (patterns) et sélecteurs de données (data structure selectors) de SML sont bâtis au moyen de la construction `case` (et également `if`).

```
fun append(x,y) = case x of nil => y | h::t => h::append(t,y)
```

Le corps de la fonction `append3` subit les transformations suivantes.

```
s0)  append(append(x,y),z)
s1)  append(case x of nil  => y
                    | h::t => h::append(t,y),z)
s2)  case x of nil  => append(y,z)
             | h::t => append(h::append(t,y),z)
s3)  case x of nil  => append(y,z)
             | h::t => case (h::append(t,y)) of nil    => z
                                              | h'::t' => h'::append(t',z)
s4)  case x of nil  => append(y,z)
             | h::t => h::append(append(t,y),z)
s5)  case x of nil  => append(y,z)
             | h::t => h::append3(t,y,z)
```

L'étape (s1) correspond au dépliage de l'occurrence interne de la fonction `append`. L'étape suivante (s2) est typique de la supercompilation : on remonte la conditionnelle `case` à l'extérieur de l'appel fonctionnel `append` ; cette transformation porte le nom de « *driving* » (cf. §1.4.3). On peut ensuite (s3) déplier le second appel de `append`. Il apparaît ainsi un test sur `h::append(t,y)`, que l'on peut résoudre immédiatement par une transformation paresseuse (s4). Enfin un pliage un peu plus général que celui employé en spécialisation conduit à la version finale (s5).

Style « définition clausale ».

On peut suivre sensiblement le même schéma si l'on utilise des définitions clausales. Dans ce style de programmation, la fonction `append` s'écrit ainsi :

```
fun append(nil ,y) = y
  | append(h::t,y) = h::append(t,y)
```

La définition de `append3` est transformée ainsi :

```
c0)  fun append3(x,y,z) = append(append(x,y),z)
c1)  fun append3(nil ,y,z) = append(y,z)
       | append3(h::t,y,z) = append(h::append(t,y),z)
c2)  fun append3(nil ,y,z) = append(y,z)
       | append3(h::t,y,z) = h::append(append(t,y),z)
c3)  fun append3(nil ,y,z) = append(y,z)
       | append3(h::t,y,z) = h::append3(t,y,z)
```

Le dépliage de la définition du `append` le plus interne dans (c0) propage des clauses dans la définition (c1) de `append3` : dépliage et driving sont simultanés. Le dépliage du second `append` et la transformation paresseuse sont également simultanés (c2). Le résultat final est la forme clausale de la définition précédente (s5), que l'on atteint en un peu moins d'étapes qu'avec les simples `case`.

Style « primitives ».

A l'opposé, conserver un style LISPien, en ayant recours à des primitives plutôt qu'à des sélecteurs de données, augmente un peu le nombre de transformations intermédiaires. Nous n'explicitons ici que les points importants des transformations du corps de `append3`.

```
fun append(x,y) = if (null x) then y else (hd x)::append(tl x,y)

p0)  append(append(x,y),z)
p1)  append(if (null x) then y else (hd x)::append(tl x,y),z)
p2)  if (null x) then append(y,z) else append((hd x)::append(tl x,y),z)
p3)  if ... else ( if null((hd x)::append(tl x,y)) then ... else ... )
p4)  if ... else ( if false then ... else hd(...)::append(tl(...),z) )
p5)  if ... else hd(hd x::append(tl x,y))::append(tl(hd x::append(tl x,y)),z)
p6)  if ... else (hd x)::append(append(tl x,y),z)
p7)  if (null x) then append(y,z) else (hd x)::append3(tl x,y,z)
```

En (p1) on a déplié le `append` le plus interne de (p0). L'étape (p2) correspond au driving. En (p3) le `append` externe dans la branche `else` a été déplié. L'expression résultante est simplifiée en (p4,p5,p6) par évaluation, paresseuse ou non. La configuration de `append3` est finalement repliée en (p7). Au style près, le résultat est identique aux précédents : la liste x est copiée en tête de `append(`y`,`z`)`.

1.4.2 Applications de la supercompilation.

La supercompilation a pour premier objectif l'optimisation. Turchin envisage également [Tur86] des applications pour les bases de données et les systèmes experts. Il considère en fait la supercompilation comme un outil de résolution de problèmes et de preuve de théorème. Les techniques de la supercompilation ont également été employées pour l'inversion de fonctions [Rom91].

1.4.3 Analyse de la supercompilation.

Langage et sémantique.

La supercompilation a été initalement présentée dans le cadre d'un langage fonctionnel peu commun, REFAL[5] [Tur86]. Comme le suggère les exemples ci-dessus, il est généralement plus simple, mais pas indispensable, de faire de la supercompilation dans un langage qui dispose de motifs et de définitions clausales. Parce que la programmation logique est dotée d'un mécanisme voisin et que les évaluateurs partiels pour PROLOG [Has88, Sah91a] font usage du redémarrage généralisé (voir plus bas) en jouant sur l'ambiguïté entre variable logique et variable représentant un objet dynamique inconnu, ils se situent à une position intermédiaire entre spécialisation et supercompilation.

Comme le remarque Mogensen [Mog89a], la *déforestation* [Wal88] est une instance de supercompilation sur un langage plus simple, ce qui facilite par ailleurs le problème de la terminaison des transformations.

Transformations.

Comme la spécialisation, la supercompilation utilise dépliage (étapes s1, s3, p1, p3 de la section 1.4.1) et évaluation (p5). Son pliage est en revanche étendu à tout type d'expression (s5, c3, p7) grâce à la transformation de *généralisation*. Elle fait un emploi indispensable de transformations paresseuses (s4, c2, p4, p6) et introduit de plus la notion de transformation par *driving* (s2, c1, p2).

Pliage, dépliage et généralisation. Les possibilités de pliage et de dépliage sont peu nuancées en spécialisation de base. Une expression comme `f(a,b,c+g(v))` où `a`, `b` et `c` sont des constantes, `g` une fonction, et `v` une variable *dynamique*, n'offre que deux éventualités : soit un pliage $\mathtt{f}_{\mathtt{a,b}}\mathtt{(c+g(v))}$ sur une fonction spécialisée $\mathtt{f}_{\mathtt{a,b}}$, soit un dépliage de la définition de `f`. On peut faire apparaître ces alternatives comme deux manières différentes d'écrire[6] l'expression initiale :

$$\mathtt{f(a,b,c+g(v))} = \lambda z.\mathtt{f(a,b,}z\mathtt{)}\ \mathtt{(c+g(v))} = \lambda xyz.\mathtt{f(}x\mathtt{,}y\mathtt{,}z\mathtt{)}\ \mathtt{(a,b,c+g(v))}$$

La première permet l'identification à $\mathtt{f}_{\mathtt{a,b}}$, et la seconde à `f`.

La *généralisation* n'est pas une transformation proprement dite. C'est elle qui énumère toute la gamme des appels fonctionnels qui sont à la fois plus généraux que `f(a,b,c+g(v))` et plus particuliers que $\mathtt{f(}x\mathtt{,}y\mathtt{,}z\mathtt{)}$. Entre ces deux extrêmes, on trouve par exemple $\mathtt{f}_{\mathtt{a}} = \lambda yz.\mathtt{f(a,}y\mathtt{,}z\mathtt{)}$ et $\mathtt{f}_{\mathtt{b}} = \lambda xz.\mathtt{f(}x\mathtt{,b,}z\mathtt{)}$, qui peuvent être utilisés pour des spécialisations moins fines que $\mathtt{f}_{a,b}$. Les autres configurations sont spécifiques à la supercompilation, qui ne fait pas de distinction entre constantes et expressions dynamiques, et autorise le pliage de tout type d'expression. Par exemple, la configuration $\mathtt{f(}x\mathtt{,b,}z_1\mathtt{+}z_2\mathtt{)}$ permet de plier l'expression `f(a,b,c+g(v))` en `h(b,c,g(v))` à l'aide de la fonction spécialisée $\mathtt{h} = \lambda x z_1 z_2.\mathtt{f(}x\mathtt{,b,}z_1\mathtt{+}z_2\mathtt{)}$. Au total, l'ensemble des configurations possibles correspond aux choix indépendants (x ou `a`), (y ou `b`) et (z ou $z_1\mathtt{+}z_2$ ou $\mathtt{c+}z_2$ ou $z_1\mathtt{+g(}z_2\mathtt{)}$ ou $\mathtt{c+g(}z\mathtt{)}$).

[5] Comme en HOPE ou en ML, une fonction REFAL a une définition clausale. Cependant, au lieu d'un simple recouvrement (pattern matching), appliquer une fonction demande l'*unification* avec une des têtes de clause. Le type de donnée essentiel de REFAL repose sur des listes associatives ; les motifs sont linéaires, et à certaines variables peut correspondre une sous-liste.

[6] Dans l'ensemble de ce document, nous employons la notation « λ » uniquement comme une facilité d'écriture des fonctions.

La généralisation est notamment utilisée pour assurer la terminaison de l'algorithme de supercompilation, car toute expression n'a qu'un nombre fini d'expressions plus générales. C'est elle également qui précède l'introduction des nouvelles définitions.

Dès lors que le pliage n'est plus restreint aux seules fonctions spécialisées sur des constantes, il devient l'exact opposé du dépliage. Il permet ainsi de conclure dans chacun des exemples (s5, c3, p7). Toutes ces transformations sont locales.

Driving. Le *driving* consiste à propager à travers le programme les sélecteurs de structure de données. Par exemple, pour le `case` de SML,

$$\texttt{f(case } e \texttt{ of } p_1 \texttt{ => } e_1 \texttt{ |...| } p_n \texttt{ => } e_n\texttt{)} \;\rightarrow\; \texttt{case } e \texttt{ of } p_1 \texttt{ => f(}e_1\texttt{) |...| } p_n \texttt{ => f(}e_n\texttt{)}$$

Son but est d'amener des sélecteurs de données au contact de contructeurs de données afin qu'ils puissent être annulés par transformation paresseuse, à la manière des étapes (s4, c2, p4, p6) de la section 1.4.1. Si l'on utilise des définitions clausales, dépliage et driving sont simultanés, comme le sont également dépliage et transformations paresseuses. Cela explique la propension, en supercompilation, pour les langages comportant des définitions clausales.

Nous avons en fait déjà rencontré cette transformation[7] dans une des extensions de la spécialisation, à des valeurs conditionellement statiques : $\texttt{f(if } a \texttt{ then } b \texttt{ else } c\texttt{)} \rightarrow \texttt{if } a \texttt{ then f(}b\texttt{) else f(}c\texttt{)}$ (cf. §1.3.5). C'est également une forme de driving que l'on utilise dans la compilation de motifs (patterns), afin d'aplatir les conditionnelles emboitées [Car84, Aug85] : $\texttt{if (if } x \texttt{ then } y \texttt{ else } z\texttt{) then } a \texttt{ else } b \rightarrow \texttt{if } x \texttt{ then (if } y \texttt{ then } a \texttt{ else } b\texttt{) else (if } z \texttt{ then } a \texttt{ else } b\texttt{)}$.

Propagation des contraintes de motif. Le driving de clauses procède par unification. Dans l'exemple (c1), nous avons unifié `append(x,y)` avec, successivement, `append(nil,y)` et `append(h::t,y)`. Les substitutions résultantes ont ensuite été appliquées à l'ensemble de la définition de `append3`. C'est pourquoi le driving de clauses est plus puissant que le driving de sélecteurs de données dans le cas de fonctions non-linéaires.

Par exemple, dans la fonction `lminmax` suivante, qui calcule la paire constituée du plus petit et du plus grand élément d'une liste, la variable `l` apparaît plusieurs fois (les variables `x`, `y` et `z` n'apparaissent qu'une fois dans la définition de `append3`).

```
fun lminmax(l,(a,b)) = ( lmin(l,a) , lmax(l,b) )
```

Les valeurs `a` et `b` sont les valeurs retournées dans le cas où `l` est vide.

Considérons une définition clausale des fonctions `lmin` et `lmax`, qui calculent respectivement le $\min$ et le $\max$ d'une liste.

```
fun lmin([],a) = a | lmin(x::l,a) = lmin(l,min(a,x))
fun lmax([],b) = a | lmax(x::l,b) = lmax(l,max(b,x))
```

La supercompilation de `lminmax` se traduit par :

[7]La construction `if` est un cas particulier de `case` sur les expressions booléennes : « $\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3$ » est équivalent à « $\texttt{case } exp_1 \texttt{ of true => } exp_2 \texttt{ | false => } exp_3$ ».

```
pc0)  fun lminmax(l,(a,b))    = ( lmin(l,a) , lmax(l,b) )
pc1)  fun lminmax([],(a,b))   = ( a , lmax([],b) )
        | lminmax(x::l,(a,b)) = ( lmin(l,min(a,x)),lmax(x::l,b) )
pc2)  fun lminmax([],(a,b))   = ( a , b )
        | lminmax(x::l,(a,b)) = ( lmin(l,min(a,x)) , lmax(l,max(b,x)) )
pc3)  fun lminmax([],(a,b))   = ( a , b )
        | lminmax(x::l,(a,b)) = lminmax(l,(min(a,x),max(b,x)))
```

En (pc1), on a déplié `lmin` ; le driving est simultané : les substitutions $\{$`l` $\mapsto$ `nil`$\}$ et $\{$`l` $\mapsto$ `x::l`$\}$ sont appliquées dans chaque clause au corps de `lminmax`. En (pc2), on a évalué paresseusement chaque branche, avec dépliage et driving de `lmax`. En (pc3), on a effectué un pliage selon la définition initiale de `lminmax`. La fonction `lminmax` résultante est meilleure car elle ne parcourt qu'une fois la liste `l` au lieu de deux.

En revanche, si les fonctions `lmin` et `lmax` sont définies par des sélecteurs de données,

```
fun lmin(l,a) = case l of [] => a | x::l => lmin(l,min(a,x))
fun lmax(l,b) = case l of [] => b | x::l => lmax(l,max(b,x))
```

la supercompilation conduit à :

```
ps0)  fun lminmax(l,(a,b)) = ( lmin(l,a) , lmax(l,b) )
ps1)  fun lminmax(l,(a,b)) =
        ( case l of [] => a | x1::l1 => lmin(l1,min(a,x1)) ,
          case l of [] => b | x2::l2 => lmax(l2,max(b,x2)))
ps2)  fun lminmax(l,(a,b)) = case l of
          [] =>
            (case l of []     => ( a , b )
                     | x2::l2 => ( a , lmax(l2,max(b,x2))) )
        | x1::l1 =>
            (case l of []     => ( lmin(l1,min(a,x1)) , b )
                     | x2::l2 => ( lmin(l1,min(a,x1)) , lmax(l2,max(b,x2))) )
```

En (ps1) on a déplié les fonctions `lmin` et `lmax`. En (ps2), les sélecteurs `case` ont été sortis par driving de la construction de la paire résultat. Non seulement il subsiste des tests inutiles (la même recherche de motif est appliquée successivement deux fois sur la variable `l`), mais surtout on n'a pas pu replier la définition (ps0) de `lminmax` parce que les variables `l1` et `l2` sont a priori différentes. La fonction `lminmax` résultante parcourt deux fois la liste `l`.

Si un sélecteur de données opère sur une variable, comme c'est le cas ci-dessus pour `l`, la *propagation des contraintes de motifs* permet d'obtenir un résultat similaire au cas d'une définition clausale. Elle consiste à appliquer alors dans chaque branche du sélecteur les substitutions issues de la recherche de motif (pattern matching) : dans l'expression `case` var `of` $motif$ `=>` exp, la recherche de motif entre var et $motif$ produit la substitution $\{var \mapsto motif\}$, que l'on applique à exp pour former la nouvelle expression `case` var `of` $motif$ `=>` $exp[var/motif]$. Dans le cas de (ps2) ci-dessus, la variable var correspondante est le `l` le plus à l'extérieur, et les motifs sont `[]` et `x1::l1`.

```
ps3)  fun lminmax(l,(a,b)) = case l of
          [] =>
            (case [] of []     => (a,b)
                      | x2::l2 => (a,lmax(l2,max(b,x2))))
```

```
          | x1::l1 =>
              (case x1::l1 of []      => (lmin(l1,min(a,x1)),b)
                             | x2::l2 => (lmin(l1,min(a,x1)),lmax(l1,max(b,x1))))
ps4)  fun lminmax(l,(a,b)) = case l of
            []     => (a,b)
          | x2::l2 => lminmax(l2,(min(a,x2),max(b,x2)))
```

En (ps3) on a appliqué les substitutions $\{\texttt{l} \mapsto \texttt{nil}\}$ et $\{\texttt{l} \mapsto \texttt{x1::l1}\}$ dans chaque branche, puis à nouveau la substitution $\{\texttt{x2} \mapsto \texttt{x1}, \texttt{l2} \mapsto \texttt{l1}\}$. Après évaluation et pliage de `lminmax`, désormais possible, on obtient en (ps4) une fonction similaire à celle obtenue en (pc3), qui ne parcourt qu'une seule fois la liste `l`.

Notons que le driving de définitions clausales est a fortiori également plus puissant que le driving de conditionelles dans un style « primitives », puisque le `if` est un cas particulier de sélecteur de données. Pour obtenir un résultat analogue, il faut en ce cas avoir recours à l'évaluation partielle généralisée (cf. §1.5.2).

Correction.

Comme en spécialisation, on limite souvent l'emploi du pliage aux nouvelles définitions afin de garantir sa correction. Alors que les transformations paresseuses qui le motivent sont complètes, le driving lui-même est une transformation valide. Il est complet si les fonctions à travers lesquelles on remonte les sélecteurs de données sont strictes (si elles sont indéfinies lorsque l'un de leurs arguments l'est aussi). En revanche, la généralisation est toujours correcte.

Algorithme.

Afin d'assurer la terminaison, la supercompilation utilise un mécanisme de *redémarrage généralisé* (generalized restart). Il simule l'évaluation du programme en faisant des calculs explicites lorsque les données sont connues, et propage les sélecteurs de données lorsqu'elles sont inconnues. Un historique de chaque étape est conservé et un test strict garantit que cet historique reste fini. Il compare pour cela l'expression courante à chacune des expressions considérées lors des étapes précédentes. En cas de risque de boucle par rapport à l'une des étapes antérieures, on reprend totalement le calcul à ce niveau, avec une configuration plus générale. Un ordre bien fondé relie ces configurations de généralité croissante et assure la finitude du nombre de configurations énumérées. Cet algorithme est repris plus en détail dans le chapitre 6.

Comparé aux algorithmes de spécialisation ordinaires, le mécanisme de redémarrage généralisé est puissant mais aussi plus coûteux en espace et en temps. Bien qu'il ne soit pas spécifiquement lié à la supercompilation, il est plus rare en spécialisation [Has88, Sah91a, WCRS91]. L'objectif reste d'obtenir les versions les plus spécialisées possibles des fonctions du programme, c'est-à-dire les moins générales. En PROLOG, l'expression courante est un atome et la configuration de généralité minimale est obtenue en calculant une anti-unification [Plo70] de cet atome avec l'atome correspondant à l'étape antérieure [Has88, Sah91a, WCRS91]. L'anti-unification s'étend aux listes associatives de REFAL [Tur88].

Terminaison.

Comme en spécialisation, un moyen simple d'assurer la terminaison est de borner le nombre d'appels récursifs d'une même fonction et d'imposer une décroissance de taille des termes (cf. §1.3.4). Seul le supercompilateur pour REFAL de Turchin utilise un test original. Toutefois, la présentation de ce critère d'arrêt dans [Tur88] est étroitement lié à l'algorithme de supercompilation et à la nature du langage.

Par ailleurs, les démonstrations de la terminaison de l'algorithme de redémarrage généralisé [Tur88, Has88, Sah91a, WCRS91] semblent négliger des explications sur la nature des redémarrages possibles. En effet, la justification de la terminaison se ramène à ceci : « toute expression n'a qu'un nombre fini d'expressions plus générales, donc l'algorithme se termine ». Ce raisonnement semble faire l'*hypothèse implicite* qu'une fois qu'un redémarrage à eu lieu à une certaine étape, tous les redémarrages suivants le sont aussi relativement à cette même étape. Or ce n'est pas nécessairement le cas. Nous pourrions avoir une suite infinie de redémarrage successifs à des étapes d'indices croissants. Rien n'interdit non plus un second redémarrage à une étape antérieure. Cette justification incomplète sera reprise à la section §6.1, où nous démontrerons plus rigoureusement la terminaison de l'algorithme.

Objet des transformations.

Outre son usage pour la terminaison, la généralisation permet de donner toute leur puissance au pliage, au dépliage, et à l'introduction de nouvelles définitions, qui seuls expriment le gain en performance de la transformation.

De la même manière, le driving n'influe pas directement sur la performance des programmes ; son intérêt est dans les transformations paresseuses qu'il rend possibles.

1.5 Évaluation partielle généralisée.

Futamura et Nogi ont proposé la notion d'*évaluation partielle généralisée* (generalized partial computation, ou GPC) [FN88, Fut88, FNT91], dont le principe est d'exploiter au maximum la sémantique et la structure logique des programmes. L'optimisation est assistée par un *système de preuve formelle* (theorem prover) ou de *calcul formel* (computer algebra) qui axiomatise les types de données abstraits et les propriétés des fonctions primitives.

1.5.1 Principes de l'évaluation partielle généralisée.

L'*évaluation partielle généralisée* conserve intégralement la trace des résultats conditionnels sous forme d'une information qui est propagée dans chaque branche de la condition. Par exemple, dans le cas d'une expression `if` c `then` x `else` y, l'information ρ progagée dans les branches x et y est respectivement c et $\neg c$. Cette information est utilisée pour décider statiquement de la valeur d'expressions contenues dans x et y, et notamment celles d'autres conditions : un système de preuve formelle tente de valider $\rho \vdash c'$ et $\rho \vdash \neg c'$, où ρ est le *contexte* d'évaluation du test c', et $\rho \vdash c'$ l'expression que les hypothèses ρ permettent de prouver la proposition logique c'. Le système de preuve peut échouer, soit par manque d'information si les données ne sont pas suffisantes, soit du fait de la difficulté intrinsèque des preuves de $\rho \vdash c'$ et $\rho \vdash \neg c'$.

À la différence d'un environnement, qui contient ordinairement les valeurs des variables et est réduit à de simples liaisons comme $\{x \mapsto 3\}$, le contexte ρ comporte des formules logiques portant sur toutes les constructions du langage, comme par exemple $\{x = 3, y > 5, \neg null(z) \wedge null(cdr(z))\}$; dans un tel contexte, le test `y - x > 4` est par exemple toujours vrai. Nous avons rencontré ce type d'analyse dans l'*évaluation partielle paramétrable*, extension de la spécialisation (cf. §1.3.5).

L'évaluation partielle généralisée ne s'intéresse pas uniquement aux tests, c'est-à-dire aux expressions à valeur booléenne, mais à toutes les expressions du langage, types de données comme expressions numériques. La *logique sous-jacente* (underlying logic) opère alors sur l'ensemble de la sémantique, depuis les axiomes des types abstraits aux propriétés algébriques des fonctions primitives. Par exemple, il est possible de substituer la constante `4` à l'expression `x*x` dans l'environnement $\{x = 2 \ \lor\ x = -2\}$. Dans le cas d'expressions numériques, c'est davantage un *système de calcul formel* qu'un système de preuve formelle qui réalise les *simplifications sémantiques*.

Même sans propagation de contraintes, l'évaluation partielle généralisée peut conclure là où la spécialisation et la supercompilation ordinaire échouent : une expression statique comme `or(x,not(x))` est simplifiée en `true`.

1.5.2 Exemples d'évaluation partielle généralisée.

L'évaluation partielle généralisée permet par exemple l'optimisation suivante.

```
if x > 2 then (if 3*x < 5 then A else B) else C   →   if x > 2 then B else C
```

La contrainte `x > 2` est telle que la condition `3*x < 5` est toujours fausse. Le texte brut d'un tel programme peut sembler rare en pratique ; il faut imaginer que les contraintes se propagent dans les appels fonctionnels. Le cheval de bataille de l'évaluation partielle généralisée — il fut aussi celui de l'interprétation abstraite — est la *fonction 91* de McCarthy.

```
fun MC91 x = if (x > 100) then x-10 else MC91(MC91(x+11))
```

Elle est simplifiée après dépliage et propagation des contraintes [FN88, FOF88] en

```
fun MC91 x = if (x > 100) then x-10 else 91
```

Cet exemple, relativement spectaculaire, est en fait très spécifique.

Nous pouvons reprendre l'exemple de `lminmax` donné comme illustration de la propagation des contraintes de motif (cf. §1.4.3). Supposons `lmin` et `lmax` définies à l'aide de primitives :

```
fun lmin(l,a) = if null(l) then a else lmin(tl(l),min(a,hd(l)))
fun lmax(l,b) = if null(l) then b else lmax(tl(l),max(b,hd(l)))
```

Par dépliage (pp1), driving (pp2), et pliage (pp3), on construit :

```
pp0)  fun lminmax(l,(a,b)) = ( lmin(l,a) , lmax(l,b) )
pp1)  fun lminmax(l,(a,b)) =
        ( if null(l) then a else lmin(tl(l),min(a,hd(l))) ,
          if null(l) then b else lmax(tl(l),max(b,hd(l))))
pp2)  fun lminmax(l,(a,b)) =
        if null(l) then
          if null(l) then ( a , b ) else ( a , lmax(tl(l),max(b,hd(l))) )
        else
```

```
              if null(l) then
                ( lmin(tl(l),min(a,hd(l))) , b )
              else
                ( lmin(tl(l),min(a,hd(l))) , lmax(tl(l),max(b,hd(l))) )
pp3)  fun lminmax(l,(a,b)) =
            if null(l) then
              if null(l) then ( a , b ) else ( a , lmax(tl(l),max(b,hd(l))) )
            else
              if null(l) then
                ( lmin(tl(l),min(a,hd(l))) , b )
              else
                lminmax(tl(l),(min(a,hd(l)),max(b,hd(l))))
```

Même si la fonction `lminmax` a pu être repliée, il subsiste, comme dans le cas des sélecteurs de données (cf. §1.4.3), des tests inutiles. Mais sans recherche de motif, il n'y a pas de substitution applicable à chaque branche des conditionnelles afin de propager les contraintes de motif. Grâce à l'évaluation partielle généralisée, on propage dans ces branches les informations $\{null(l)\}$ et $\{\neg null(l)\}$; les tests `null(l)` suivants peuvent être résolus et l'on obtient (pp4).

```
pp4) fun lminmax(l,(a,b)) =
       if null(l) then
         ( a , b )
       else
         lminmax(tl(l),(min(a,hd(l)),max(b,hd(l))))
```

La fonction `lminmax` en (pp4) a, au style près, un comportement similaire aux fonctions obtenues en (pc3) et en (ps4).

1.5.3 Analyse de l'évaluation partielle généralisée.

Langages et sémantiques.

L'évaluation partielle généralisée a été présentée à l'origine sur un noyau LISP [FN88, Fut88, FNT91]. La méthode a ensuite été appliquée à la programmation logique avec contraintes, qui intègre naturellement programmes, contraintes et preuves : clauses de Horn avec garde (guarded Horn clauses) [FOF88] et contraintes booléennes [HS91]. Dans le cas fonctionnel, elle a été étendue à l'évaluation paresseuse [Tak91].

Il existe encore peu d'implémentations d'évaluateur partiel généralisé ; les systèmes de preuve formelle et l'axiomatisation des types de données abstraits et des propriétés des fonctions primitives sont souvent hypothétiques. Ce problème ne se pose pas pour la programmation logique avec contrainte, qui contient en quelque sorte son propre résolveur (CLP(BOOL), etc.) [FOF88].

Transformations.

La propagation du contexte d'évaluation n'est pas une transformation en soi. La seule opération sur les programmes de l'évaluation partielle généralisée est la *simplification sémantique*, qui étend l'évaluation d'expressions statiques car elle opère sur des expressions qui ne sont pas nécessairement connnues en totalité. Bien que l'information se propage dans tout le programme, la transformation elle-même est locale.

Correction.

La correction de la résolution des formules logiques et sémantiques repose totalement sur l'axiomatisation fournie aux systèmes de preuve et de calcul formel, qui doit être *compatible* avec la sémantique du langage [FN88]. De plus, il ne doit pas être possible de prouver des « théorèmes paresseux ». Par exemple, si la fonction `f` est définie par `fun f(x) = 3+abs(f(x))`, et donc boucle sur toute donnée d'entrée, on ne doit pas substituer $true$ à l'expression `f(x) > 1` sous prétexte que $3 + |x| > 1$ pour tout réel x.

Algorithme.

À ses débuts, l'évaluation partielle généralisée s'est préoccupée principalement des problèmes de propagation et de résolution de contraintes. Pas d'algorithme donc, mais « a highly nondeterministic procedure » [FN88, §5]. Les études suivantes ont proposé des algorithmes dictés en partie par la nature du langage [FOF88, Tak91, HS91].

Terminaison.

L'évaluation partielle généralisée n'apporte ni problème nouveau, ni solution nouvelle, en matière de terminaison : « finding practical methods for automatic termination is an interesting research problem » [FN88, §3]. Elle emploie donc les mêmes techniques de décroissance que celles utilisées en spécialisation et en supercompilation. Certains « algorithmes » avouent ne pas toujours terminer [Tak91]. D'autres permettent à l'utilisateur de spécifier abstraitement un critère d'arrêt [FOF88, HS91].

Objet des transformations.

Le gain de la simplification sémantique est du même type que celui de l'évaluation d'expression statique. On économise des calculs dynamiques en les réalisant statiquement. De plus, à l'image du driving, cette simplification crée de nouvelles expressions réductibles (redices) qui permettent de faire davantage d'optimisations. La puissance de la simplification sémantique réside dans celle du système de preuve ou de calcul formel, et dans l'axiomatisation des types de données abstraits et des propriétés des fonctions primitives.

L'automatisation de l'évaluation partielle généralisée est liée dans une certaine mesure aux résultats de démonstration automatique. Par exemple, la décision du calcul propositionnel permet systématiquement de simplifier ou non une classe de conditions booléennes.

1.6 Bilan comparatif.

Nous dressons à présent un bilan comparatif de ces approches, d'une part en terme de puissance des optimisations (quelles transformations sont ou ne sont pas pas possibles), et d'autre part en suivant à nouveau la méthodologie proposée à la section §1.1.5. Ce bilan motive les sujets abordés dans les chapitres suivants.

1.6.1 Puissance comparée des évaluations partielles.

Nous examinons sur une même expression la puissance des évaluations partielles décrites dans les sections précédentes, et plus précisément l'emploi qu'elles font des informations d'ordre sémantique disponibles.

Évaluation.

Notons $\boldsymbol{eval}[\![exp]\!]\ \rho$ l'*évaluation* de l'expression exp dans l'environnement ρ. Cet environnement contient les valeurs courantes des variables, connues ou inconnues (une valeur indéfinie est notée $\perp$). Considérons l'expression conditionnelle « $\texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y$ » :

$$\begin{array}{l}\boldsymbol{eval}[\![\texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y]\!]\ \rho =\\ \qquad \begin{array}{lll}\text{si}\ \boldsymbol{eval}[\![c]\!]\ \rho = \mathit{true} & \text{alors}\ \boldsymbol{eval}[\![x]\!]\ \rho\\ \text{si}\ \boldsymbol{eval}[\![c]\!]\ \rho = \mathit{false} & \text{alors}\ \boldsymbol{eval}[\![y]\!]\ \rho\\ \text{si}\ \boldsymbol{eval}[\![c]\!]\ \rho = \perp & \text{alors}\ \perp\end{array}\end{array}$$

L'évaluation pure (« totale » par opposition à « partielle ») n'est définie que sur les termes entierement connus.

Spécialisation.

Au lieu de rester bloqué sur les parties inconnues, la *spécialisation* $\boldsymbol{spec}[\![exp]\!]\ \rho$ effectue les calculs sur les expressions statiques et laisse le reste inchangé.

$$\begin{array}{l}\boldsymbol{spec}[\![\texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y]\!]\ \rho =\\ \qquad \begin{array}{ll}\text{si}\ \boldsymbol{spec}[\![c]\!]\ \rho = \mathit{true} & \text{alors}\ \boldsymbol{spec}[\![x]\!]\ \rho\\ \text{si}\ \boldsymbol{spec}[\![c]\!]\ \rho = \mathit{false} & \text{alors}\ \boldsymbol{spec}[\![y]\!]\ \rho\\ \multicolumn{2}{l}{\text{sinon}\ \ \texttt{if}\ (\boldsymbol{spec}[\![c]\!]\ \rho)\ \texttt{then}\ (\boldsymbol{spec}[\![x]\!]\ \rho)\ \texttt{else}\ (\boldsymbol{spec}[\![y]\!]\ \rho)}\end{array}\end{array}$$

Cependant, toutes les informations disponibles ne sont pas exploitées. Dans les deux branches du `if` résultant, $\boldsymbol{spec}[\![x]\!]\ \rho$ et $\boldsymbol{spec}[\![y]\!]\ \rho$, on ne relève pas trace du fait que la condition c est respectivement vraie ou fausse.

Supercompilation.

Dans le cas particulier où le test est fait sur une variable, la *supercompilation* par propagation des contraintes de motif d'un selecteur de données, ou par driving d'une définition sous forme fonctionnelle clausale, diffuse sous forme d'une substitution une information booléenne le long de ces branches (cf. §1.4.3).

$$\begin{array}{l}\boldsymbol{scomp}[\![\texttt{if}\ var\ \texttt{then}\ x\ \texttt{else}\ y]\!]\ \rho =\\ \qquad \begin{array}{ll}\text{si}\ \rho(var) = \mathit{true} & \text{alors}\ \boldsymbol{scomp}[\![x]\!]\ \rho\\ \text{si}\ \rho(var) = \mathit{false} & \text{alors}\ \boldsymbol{scomp}[\![y]\!]\ \rho\\ \multicolumn{2}{l}{\text{sinon}\ \ \texttt{if}\ var\ \texttt{then}\ (\boldsymbol{scomp}[\![x.\{var \mapsto \texttt{true}\}]\!]\ \rho)\ \texttt{else}\ (\boldsymbol{scomp}[\![y.\{var \mapsto \texttt{false}\}]\!]\ \rho)}\end{array}\end{array}$$

Plus généralement, elle propage des contraintes de constructeurs de types de données dans chaque branche d'un `case` en substituant les occurrences de la variable par le motif associé. La supercompilation dispose également d'un pliage plus puissant que celui de la spécialisation, et peut évaluer davantage d'expressions statiques grâce au driving.

Évaluation partielle généralisée.

Alors que la supercompilation peut propager une information structurelle (syntaxique), et ce uniquement dans le cas d'un test sur une variable, l'*évaluation partielle généralisée* $\boldsymbol{epg}[\![exp]\!]\ \rho$ propage pout tout test une information d'ordre sémantique. Dans le cas d'une conditionnelle, le contexte ρ est enrichi dans chacune des deux branches des informations respectives c et $\neg c$.

$$
\begin{array}{l}
\boldsymbol{epg}[\![\texttt{if } c \texttt{ then } x \texttt{ else } y]\!]\ \rho = \\
\quad \text{si } \rho \vdash c \quad \text{alors} \quad \boldsymbol{epg}[\![x]\!]\ \rho \\
\quad \text{si } \rho \vdash \neg c \quad \text{alors} \quad \boldsymbol{epg}[\![y]\!]\ \rho \\
\quad \text{sinon} \quad \texttt{if } (\boldsymbol{epg}[\![c]\!]\ \rho) \texttt{ then } (\boldsymbol{epg}[\![x]\!]\ (\rho \wedge c)) \texttt{ else } (\boldsymbol{epg}[\![y]\!]\ (\rho \wedge \neg c))
\end{array}
$$

La notation $\rho \vdash c$ signifie que l'on peut prouver c à partir des informations ρ. Le cas « sinon » correspond à un échec du système de preuve, qui n'a pu décider la vérité de c.

Notons que la supercompilation peut parfois modéliser des contraintes sémantiques. Par exemple, la contrainte $n < 2$ peut être représentée à l'aide constructeurs de données par $n = \texttt{S(S(S(}m\texttt{)))}$ (représentation unaire des entiers positifs), contraintes qu'elle exploite ensuite par recherche de motifs. Néanmoins, elle reste impuissante devant d'une contrainte plus complexe comme $premier(n)$.

1.6.2 Analyse comparative des évaluations partielles.

Malgré leurs différences, toutes ces méthodes tendent vers un même procédé. Les extensions de la spécialisation empiètent sur la supercompilation (cf. §1.3.5) et l'évaluation partielle généralisée (cf. §1.3.5). La supercompilation développe les transformations de spécialisation et annonce l'évaluation partielle généralisée (cf. §1.4.3), qui elle même ne peut exister sans les précédentes (cf. §1.5.3).

Le bilan qui suit regroupe les concepts et motive les sujets que nous abordons dans les chapitres suivants.

Langages et sémantiques. Peu de langages sont traités dans leur totalité lorsqu'ils ont des sémantiques complexes. C'est vraisemblablement parce qu'ils sont simples, uniformes et non-typés que les langages SCHEME et PROLOG sont les plus courtisés. Ils sont purement opérationnels dans le sens où ils ne comportent pas de déclarations ; ils ne nécessitent pas non plus l'écriture d'un analyseur syntaxique (parser) spécifique ; ils peuvent naturellement manipuler leurs propres programmes au même niveau que les données, ce qui facilite beaucoup l'auto-application [Lau91b].

Bien que se dégagent des idées générales, les spécificités des langages de programmation sont souvent trop marquées pour formuler des résultats universels. C'est pourquoi, à de rares exceptions près, les diverses méthodes se résument à des illustrations circonstanciées sur des langages représentatifs.

Pour être aussi général que possible, malgré ces spécificités, nous restons dans une large mesure au niveau de formalismes sémantiques. Nous employons pour cela la *sémantique naturelle* [Kah87] et les *schémas de programme récursifs* [Cou90a] (chap. 2). Nous donnons aussi des exemples concrets sur des petits langages.

Transformations. Les langages de programmation sont généralement bâtis autour de deux pôles : données et structures de contrôle. L'évaluation d'expressions statiques, le driving et la simplification sémantique sont des transformations spécifiques qui opèrent principalement sur les données ou les constructeurs de données. Au contraire, le pliage et le dépliage sont des transformations relativement universelles qui jouent sur les structures de contrôle.

Introduites initialement par Burstall et Darlington [BD77], pliage et dépliage ont fait l'objet d'études systématiques dans le cadre des schémas de programme récursifs [Cou90a]. Parce que leur formulation est algébrique et qu'ils dissocient clairement données et contrôle, les schémas de programme permettent d'établir des résultats généraux. C'est en grande partie ce formalisme que nous adoptons dans notre approche (chap. 5).

Correction. Des conditions d'utilisation précises garantissent la correction des programmes transformés. Seul le pliage semble souffrir de la plus sévère restriction ; il est limité à un emploi restreint des nouvelles définitions.

Dans le cadre des schémas de programme récursifs, Courcelle [Cou79, Cou86, Cou90a] et Kott [Kot85] donnent des conditions plus générales. Nous examinons comment étendre certains de ces résultats, pour un plus large emploi en évaluation partielle (chap. 5).

Algorithme. L'analyse de temps de liaison (BTA) est propre à la spécialisation. Insuffisante en supercompilation, elle pourrait toutefois accélérer le traitement du cas particulier des expressions statiques. En revanche, l'algorithme de redémarrage généralisé, assez peu spécifique, convient à toutes les formes d'évaluation partielle.

Nous en donnons une formalisation abstraite plus générale (§6.1), qui permet en outre d'établir clairement sa terminaison. Par ailleurs, elle permet également d'expliquer pourquoi les preuves de sa terminaison sont parfois hâtives.

Terminaison. En l'absence d'un critère d'arrêt spécifique, la terminaison repose principalement sur une borne du nombre d'appel d'une même fonction ou d'une même configuration sans qu'il y ait décroissance de taille des arguments. Les paramètres de ces bornes sont, dans une certaine mesure, empiriques.

Afin de faire le point de ces critères, nous en étudions la nature et les qualités (§6.2). Par ailleurs, nous proposons et justifions l'emploi d'un critère de terminaison particulier, voisin de celui employé par Turchin dans la supercompilation. Algébrique, il s'intègre de manière naturelle dans le formalisme des schémas de programmes.

Objet des transformations. Il n'y a pas de formulation explicite du gain des diverses transformations. En toute rigueur, on ne garantit donc pas une *optimisation*. En pratique cependant, les cas de détérioration de performance sont rares.

Nous proposons un modèle formel de performance qui permet de comparer les programmes et ainsi de justifier le gain d'une transformation (chap. 3). Grâce à ce modèle, nous pouvons formaliser une définition abstraite de l'*évaluation partielle* (chap. 4).

Conseils et heuristiques. Dans la mesure où la qualité des résultats dépend souvent du style de programmation, il est avantageux de connaître les mécanismes de base des évaluateurs partiels. Un examen a posteriori du résultat n'est pas facile car le code généré est souvent peu lisible — ce n'est pas un reproche : on ne demande pas à un compilateur de produire du binaire compréhensible.

Il faut néanmoins noter que beaucoup de recommandations sur le style de programmation pallient en fait des carences reconnues et momentanées. Les systèmes d'évaluation partielle en question projettent d'effectuer automatiquement les transformations que suggèrent informellement les recommandations.

Conclusion du chapitre.

Nous avons présenté une étude systématique et synthétique des courants majeurs de l'évaluation partielle : spécialisation (§1.3), supercompilation (§1.4), et évaluation partielle généralisée (§1.5). Nous avons mis en évidence leurs caractéristiques distinctives (transformations caractéristiques, types d'information sémantique exploités), leurs points communs (transformations communes, problèmes de correction, de terminaison), et l'écart entre les modèles et les implémentations existantes (correction, terminaison). Nous avons montré qu'à travers leurs extensions elles tendaient toutes vers un même procédé (§1.6).

Nous avons indiqué certaines faiblesses (§1.6.2), qui motivent les développements des chapitres suivants : modèle d'exécution (chap. 2), nature des optimisations (chap. 3 et 4), correction des transformations (chap. 5), algorithme et terminaison (chap. 6).

Chapitre 2

Sémantique et Exécution

La performance d'un programme n'est pas intrinsèque ; elle est liée au mode de calcul du résultat. C'est sur une description *opératoire* du comportement d'un programme, un *modèle d'exécution*, que l'on base, ensuite, un *modèle de performance*.

Parce que l'évaluation partielle traite d'optimisations de haut niveau, un modèle d'exécution est un compromis entre une implémentation concrète et une spécification abstraite. Pour approcher le détail des implémentations réelles, tout en restant dans un cadre formel de haut niveau, nous fondons notre approche, naïve et pragmatique, sur des *sémantiques opérationnelles*[1]. Nous prenons comme exemples la *sémantique naturelle* et les *schémas de programme récursifs*, qui permettent de graduer la fidélité d'un modèle d'exécution grâce à leurs facettes déclarative et opérationnelle.

Un petit langage illustre des exemples de modèles d'exécution (compilation, spécifications en sémantique naturelle et dans les schémas de programme). Il sera également le support d'exemples de mesure de performance et d'optimisation dans les chapitres qui suivent.

Organisation du chapitre.

- §2.1 Nous définissons une notion abstraite de *modèle d'exécution* qui explicite les opérations effectuées par une *machine d'exécution* afin, au chapitre suivant, d'attribuer une performance à une exécution.
- §2.2 Ces définitions sont illustrées à l'aide d'une implémentation (une compilation).
- §2.3 Un modèle d'exécution n'est jamais qu'un « modèle » de la réalité. La complexité pratique des implémentations impose des compromis sur la *fidélité* et le *degré d'abstraction* de ce modèle.
- §2.4 En tenant compte de ces contraintes, nous définissons une classe de modèles d'exécution au moyen du formalisme de *sémantique naturelle*.
- §2.5 Une classe similaire de modèles d'exécution est définie à l'aide du formalisme des *schémas de programme récursifs*.

Le chapitre suivant s'appuie sur la notion de modèle d'exécution pour formuler une notion de *modèle de performance* (chap. 3).

[1] Par *sémantique opérationnelle*, nous entendons ici une sémantique qui a une signification *opératoire*, par opposition notamment à une sémantique *déclarative*. Il ne faut pas confondre cette notion avec la *sémantique opérationnelle structurelle*, traduction de « structural operational semantics » [Plo81], plus couramment appelée en français *sémantique naturelle* [Kah87], que nous rencontrerons à la section §2.4.

Conventions et notations. Les notations et définitions mathématiques usuelles, que nous ne rappelons pas toujours dans le texte, sont données dans l'annexe N. Les notations en matière de programmation et de sémantique sont inspirées de la définition formelle de STANDARD ML [MTH90] :

- Des fontes de famille différente distinguent dans nos exemples les textes de programmes, comme « `append([1,2*3],4::nil)` », les objets algébriques de syntaxe abstraite, comme « $\mathsf{apply}(::, \mathsf{4}, \mathsf{nil})$ », et les valeurs et objets sémantiques, comme « *1::6::4::nil* ». Cela s'applique également aux identificateurs : `X`, X, X. De même, les inconnues des schémas de programme apparaissent comme « **append** », et les variables algébriques, éventuellement converties en minuscule, comme « x ».
- Nous notons en minuscules italiques, par exemple « exp », les éléments d'un ensemble (classe syntaxique ou sémantique) « Exp ».
- Nous n'explicitons pas la *syntaxe abstraite* des programmes. Elle reste néanmoins conceptuellement derrière notre discours auquel, pour plus de lisibilité, nous préférons donner l'apparence de la *syntaxe concrète*. Par exemple, « `let` dec `in` exp `end` » dénote le terme véritable « $\mathsf{let}(dec, exp)$ ».
- Dans les *règles grammaticales* et *sémantiques*, nous notons entre crochets $\langle \ldots \rangle$ les termes (simultanément) optionnels, là où nous aurions dû écrire deux règles. Nous faisons de même pour les équations. Dans tout autre contexte, ces mêmes crochets sont employés pour délimiter des suites d'objets, finies ou infinies, ou bien des n-uplets.

Afin de ne pas encombrer inutilement le texte, les démonstrations des propositions élémentaires sont rejetées en annexe, §D.2.

2.1 Formalisation de l'exécution.

La *sémantique* d'un langage de programmation L est représentée par une fonction $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ (cf. §1.1.1, 1.1.3). La fonction mathématique $\hat{L}(p) : \mathcal{D} \to \mathcal{R}$ que dénote le texte d'un programme $p \in \mathcal{P}$ est un objet statique, indépendant de toute notion d'*exécution*. Les caractéristiques dynamiques (et notamment la nature algorithmique) du programme sont seules liées à la machine (après compilation) ou à l'interprète qui réalise effectivement les calculs.

Un *modèle d'exécution* est l'*observation d'un procédé d'exécution* des programmes. De la même manière que Berry et Curien [Ber81, BC82] opposent les *algorithmes* aux *fonctions* qu'ils représentent, un modèle d'exécution distingue les programmes non pas selon leur *extension*, c'est-à-dire leur *comportement opérationnel* (le résultat net de l'exécution), mais selon la *manière* dont les résultats sont calculés. Il va à l'*opposé* du problème d'*abstraction totale* [Plo81, Sto88] qui recherche précisément des modèles où *dénotations* et *comportement opérationnel* coïncident.

Cette section contient des définitions formelles qui ne seront illustrées qu'à la section suivante, sur un exemple d'implémentation (cf. §2.2).

2.1.1 Sémantique.

Définition 2.1. (Langage de programmation)
Un *langage de programmation* L est donné par sa *sémantique*, une fonction $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ qui relie les programmes $p \in \mathcal{P}$, les données $d \in \mathcal{D}$ et les résultats $r \in \mathcal{R}$ par $L\ p\ d = r$. Pour tout $p \in \mathcal{P}$, on note :

- $Dom(p) = Dom(L(p)) = \{d \in \mathcal{D} \mid \exists r \in \mathcal{R} \quad L\ p\ d = r\}$ est le *domaine de définition* de p.

- $Im(p) = L(p, Dom(p))$ est l'*image* de p.
- $p(d)$ est l'objet *syntaxique* représentant le programme p appliqué à la donnée d.

Un langage L est de *domaines finis* ssi $Dom(p)$ est fini pour tout $p \in \mathcal{P}$.

Au sens propre, cette définition suppose qu'un programme, qui définit généralement plusieurs sous-programmes, a une notion de procédure *principale*, soit implicite à la manière de `program` en PASCAL ou de `main` en C, soit explicite avec le nommage f d'une fonction particulière de p. Si p n'est pas ambigu, on abrège $p(d)$ en $f(d)$. Lorsqu'un programme p prend plusieurs arguments, la donnée d est un n-uplet $(d_1, \ldots, d_n)$.

La déclaration d'un ensemble (une bibliothèque) de fonctions est modélisée par un n-uplet $p = (p_1, \ldots, p_n)$ de sous-programmes p_i qui définissent chacun une fonction de nom f_i. Une donnée $d = (f_i, d_i)$ appliquée à un tel programme comporte deux parties, l'une permettant de sélectionner un p_i de p, l'autre étant la donnée proprement dite ; le résultat est celui de $p_i(d_i)$.

2.1.2 Modèle d'exécution.

Un *modèle d'exécution* pour un langage L reflète une sémantique *opératoire* de L et donne un sens à l'*algorithme* dénoté par un programme. Il n'est pas spécifique au langage mais au choix d'une *machine d'exécution*.

Peuvent être considérées comme machines d'exécution : une machine concrète (microprocesseur, réseau de processeurs, etc.) ou abstraite (machine de Turing, machine SECD, WAM, etc.), un interprète de code octet ou un interprète de haut niveau, la composition d'une compilation avec une machine d'exécution (par exemple `cc`, suivi de l'exécution du binaire).

Un modèle caractérise une exécution par sa *trace*, signature abstraite du *comportement dynamique* du programme (par opposition au *comportement opérationnel*, qui est la simple observation du résultat). Dans le cas d'une machine concrète ou abstraite, cette trace est par exemple la séquence des instructions ou opérations élémentaires effectuées au cours de l'exécution.

Définition 2.2. (Modèle d'exécution)
Un *modèle d'exécution* M d'un langage $L : \mathcal{P} \times \mathcal{D} \rightarrow \mathcal{R}$ est donné par :

- un ensemble de *traces d'exécution* $\mathcal{T}$
- une *observation de l'exécution* $M : \mathcal{P} \times \mathcal{D} \rightarrow \mathcal{T}$ tq $Dom(M) = Dom(L)$

Si p n'est pas ambigu et définit une fonction $L(p)$ de nom f, on note aussi $M(f(d))$ une trace $t = M\ p\ d$.

Bien que cette formalisation puisse s'appliquer à des langages non-détermistes ($L\ p\ d = r$ est alors l'ensemble des résultats de $p(d)$, et $M\ p\ d = t$ l'ensemble des trace d'exécution), nous ne considérons en pratique que des langages séquentiels et déterministes.

2.1.3 Exécution compilée.

Si l'on dispose d'une *traduction* des programmes d'un langage source L_s vers un langage cible L_c et d'un modèle d'exécution de L_c, on en déduit un *modèle d'exécution de L_s compilé*.

Définition 2.3. (Compilation)
Une *compilation* $(T_\mathcal{P}, T_\mathcal{D}, T_\mathcal{R})$ du langage $L_s : \mathcal{P}_s \times \mathcal{D}_s \to \mathcal{R}_s$ dans le langage $L_c : \mathcal{P}_c \times \mathcal{D}_c \to \mathcal{R}_c$ est :

- une application $T_\mathcal{P} : \mathcal{P}_s \to \mathcal{P}_c$
- une application $T_\mathcal{D} : \mathcal{D}_s \to \mathcal{D}_c$
- une application $T_\mathcal{R} : \mathcal{R}_c \to \mathcal{R}_s$

telles que $L_s = T_\mathcal{R} \circ L_c \circ (T_\mathcal{P}, T_\mathcal{D})$.

La compilation correspond au diagramme suivant (voir aussi la section §1.1.2) :

$$\begin{tikzcd}
\mathcal{P}_s \arrow[d, "T_\mathcal{P}"'] & \times & \mathcal{D}_s \arrow[d, "T_\mathcal{D}"'] \arrow[r, "L_s"] & \mathcal{R}_s \\
\mathcal{P}_c & \times & \mathcal{D}_c \arrow[r, "L_c"] & \mathcal{R}_c \arrow[u, "T_\mathcal{R}"']
\end{tikzcd}$$

Définition 2.4. (Modèle d'exécution d'un langage compilé)
Soient $(T_\mathcal{P}, T_\mathcal{D}, T_\mathcal{R})$ une compilation de L_s dans L_c, et $M_c : \mathcal{P}_c \times \mathcal{D}_c \to \mathcal{T}_c$ un modèle d'exécution de L_c. Le *modèle d'exécution de L_s compilé* est la fonction $M_s : \mathcal{P}_s \times \mathcal{D}_s \to \mathcal{T}_c$ définie par $M_s = M_c \circ (T_\mathcal{P}, T_\mathcal{D})$.

C'est bien un modèle d'exécution de L_s car $Dom(M_s) = Dom(L_s)$ (cf. §D.2). Notons que les traces d'exécution par M_s sont à valeur dans $\mathcal{T}_c$ et non dans un ensemble de traces $\mathcal{T}_s$ spécifique à L_s.

2.2 Compilation.

Voici un exemple de modèle d'exécution dont la machine d'exécution est une machine physique (concrète). C'est une *implémentation* de BOOL, petit « langage jouet », volontairement réduit afin de fournir par la suite des exemples simples d'évaluation partielle *optimale*. Syntaxe et sémantique font en sorte que BOOL est un sous-ensemble de STANDARD ML. Les sections §2.4 et §2.5 donnent deux autres modèles d'exécution de ce même langage.

2.2.1 Syntaxe.

Un programme dans le langage *Bool* : Dec $\times$ (Fun $\times$ BoolVal*) $\to$ BoolVal consiste en un ensemble dec de définitions fonctionnelles mutuellement récursives. Une donnée est composée (cf. §2.1.1) d'un nom de fonction fun définie dans dec et d'un n-uplet de valeurs booléennes $v_1, \ldots, v_n \in$ BoolVal $= \{true, false\}$. Un résultat est également un booléen de BoolVal. BOOL est un langage de domaines finis (déf. 2.1).

Les expressions de BOOL sont constituées des booléens, de variables, de la conditionnelle `if` et de l'appel fonctionnel. Quelques exemples accompagnent la grammaire, figure 2.1. En accord avec les conventions données en début de chapitre, la *syntaxe abstraite* est simplement sous-entendue.

Pour simplifier, nous supposons de plus que les programmes sont « bien formés ». Pour toute définition de fonctions `fun` fun_1`(`$var_{1,1}$`, ... ,`var_{1,n_1}`) =` exp_1 `and ... and` fun_k`(`$var_{k,1}$`, ... ,`var_{k,n_k}`) =` exp_k,

- tous les identificateurs de fonction fun_i sont différents deux à deux,

$bool$	::=	`true`	`true`
		`false`	`false`
exp	::=	$bool$	`true, false`
		var	`X, Y, Z...`
		`if` exp_1 `then` exp_2 `else` exp_3	`if X then false else Y`
		$fun(\langle exp_1, \ldots, exp_n\rangle)$	`or(hop(),not(X))`
$funbind$	::=	$fun(\langle var_1, \ldots, var_n\rangle)$ `=` exp	`not(X) = if X then false else true`
$funbinds$	::=	$funbind$ $\langle$`and` $funbinds\rangle$	`id(X) = X and nor(X,Y) = not(or(X,Y))`
dec	::=	`fun` $funbinds$	`fun not(X) = if X then false else true`

Figure 2.1 : Syntaxe de BOOL

- aucun identificateur de variable $var_{i,j}$ ne porte le même nom qu'un identificateur de fonction fun_i,
- les variables qui apparaissent dans les expressions exp_i sont parmi $var_{i,1}, \ldots, var_{i,n_i}$,
- les identificateurs de fonctions qui apparaissent dans $exp_1, \ldots, exp_k$ sont parmi $fun_1, \ldots, fun_k$.

Lorsque dec n'est pas ambigu, nous abrégeons $Bool(dec, (fun, (v_1, \ldots, v_n)))$ en $Bool(fun(v_1, \ldots, v_n))$.

2.2.2 Compilation.

La figure 2.2 donne[2] une compilation très simple de BOOL vers une machine Mach comportant un indicateur conditionnel `ZF` et quatre registres : un accumulateur `AX`, un registre auxiliaire `BP`, un pointeur de pile `SP` et un pointeur d'instructions `IP` (ce pourrait presque être l'INTEL 8086 [DG82]).

La mémoire est divisée en une zone pour le code et une zone pour la pile. Cette pile est employée pour stocker des données (les arguments des fonctions) ainsi que des adresses (les adresses de retour des fonctions appelées). Elle croît vers les adresses inférieures. Une adresse tient sur deux emplacements mémoire, ceci afin d'imiter le 8086 et de donner un parfum de « cas réel ».

Voici une description informelle des *instructions* de cette *machine d'exécution* :

- `CALL` $addr$: empile l'adresse courante et poursuit l'exécution à l'adresse $addr$.
- `CMP AX,`imm : positionne l'indicateur `ZF` ssi `AX` est égal à la valeur immédiate imm.
- `HLT` : arrête la machine.
- `JMP` $addr$: poursuit l'exécution à l'adresse $addr$.
- `JZ` $addr$: poursuit l'exécution à l'adresse $addr$ ssi l'indicateur `ZF` est positionné.
- `MOV AX,`imm : range dans l'accumulateur `AX` la valeur immédiate imm.
- `MOV AX,(BP +` $depl$`)` : range dans `AX` le contenu de l'adresse `BP` augmentée d'un déplacement $depl$.
- `MOV BP,SP` : range dans `BP` la valeur de `SP`.

[2] Rappelons que les crochets $\langle \ldots \rangle$ délimitent des termes *simultanément* optionnels (cf. p. 36). Ainsi, la règle de compilation pour $\boldsymbol{comp}[\![fun(\langle var_1, \ldots, var_n\rangle) = exp]\!]$ désigne-t-elle les deux règles suivantes.

$$\boldsymbol{comp}[\![fun() = exp]\!] = fun : \boldsymbol{comp}[\![exp]\!]\,\{\}\,;\ \texttt{RET}$$

$$\boldsymbol{comp}[\![fun(var_1, \ldots, var_n) = exp]\!] = fun : \texttt{MOV BP,SP}\,;\ \boldsymbol{comp}[\![exp]\!]\,\{var_1 \mapsto 2n, \ldots, var_n \mapsto 2\}\,;\ \texttt{RET}\ 2n$$

$$
\begin{array}{lcl}
\boldsymbol{codage}[\![\texttt{true}]\!] & = & \texttt{1} \\
\boldsymbol{codage}[\![\texttt{false}]\!] & = & \texttt{0} \\
\\
\boldsymbol{comp}[\![bool]\!]\ E & = & \texttt{MOV AX,}\boldsymbol{codage}[\![bool]\!] \\
\boldsymbol{comp}[\![var]\!]\ E & = & \texttt{MOV AX,(BP+}E(var)\texttt{)} \\
\boldsymbol{comp}[\![\texttt{if }exp_1\texttt{ then }exp_2\texttt{ else }exp_3]\!]\ E & = & \text{Soient } lab \text{ et } lab' \text{ de nouvelles étiquettes,} \\
 & & \quad \boldsymbol{comp}[\![exp_1]\!]\ E \\
 & & \quad \texttt{CMP AX,0} \\
 & & \quad \texttt{JZ }lab \\
 & & \quad \boldsymbol{comp}[\![exp_2]\!]\ E \\
 & & \quad \texttt{JMP }lab' \\
 & & lab\texttt{ :} \\
 & & \quad \boldsymbol{comp}[\![exp_3]\!]\ E \\
 & & lab'\texttt{ :} \\
\boldsymbol{comp}[\![fun\texttt{(}\langle exp_1\texttt{,}\ldots\texttt{,}exp_n\rangle\texttt{)}]\!]\ E & = & \ \langle\texttt{PUSH BP} \\
 & & \quad \boldsymbol{comp}[\![exp_1]\!]\ E \\
 & & \quad \texttt{PUSH AX} \\
 & & \quad \ldots \\
 & & \quad \boldsymbol{comp}[\![exp_n]\!]\ E \\
 & & \quad \texttt{PUSH AX}\rangle \\
 & & \quad \texttt{CALL }fun \\
 & & \ \langle\texttt{POP BP}\rangle \\
\\
\boldsymbol{comp}[\![fun\texttt{(}\langle var_1\texttt{,}\ldots\texttt{,}var_n\rangle\texttt{) = }exp]\!] & = & fun\texttt{ :} \\
 & & \ \langle\texttt{MOV BP,SP}\rangle \\
 & & \quad \boldsymbol{comp}[\![exp]\!]\ \{\langle var_1 \mapsto 2n, \ldots, var_n \mapsto 2\rangle\} \\
 & & \quad \texttt{RET }\langle 2n\rangle \\
\boldsymbol{comp}[\![funbind\ \langle\texttt{and }funbinds\rangle]\!] & = & \quad \boldsymbol{comp}[\![funbind]\!] \\
 & & \ \langle\boldsymbol{comp}[\![funbinds]\!]\rangle \\
\boldsymbol{comp}[\![\texttt{fun }funbinds]\!] & = & \quad \boldsymbol{comp}[\![funbinds]\!]
\end{array}
$$

Figure 2.2 : Compilation de BOOL

- `PUSH AX` : empile la valeur de `AX` (`SP` est décrémenté de 2 et `AX` est rangé à l'adresse contenue dans `SP`).
- `PUSH BP` : empile la valeur de `BP`.
- `POP BP` : dépile le registre `BP` (`BP` recoit la valeur contenue à l'adresse `SP` et `SP` est incrémenté de 2).
- `RET` : dépile l'adresse de retour.
- `RET` $depl$: dépile l'adresse de retour et incrémente `SP` d'un déplacement $depl$.

La directive d'assemblage « lab : » repère une adresse dans le programme à l'aide d'une étiquette. On note hlt une adresse mémoire particulière de la zone code contenant l'instruction `HLT`.

Pour effectuer l'appel fonctionnel $fun(v_1, \ldots, v_n)$, on crée sur la pile un bloc d'activation (compilation record [ASU86]) contenant la valeur courante de `BP`, les paramètres et l'adresse de retour ; puis l'on poursuit l'exécution à l'adresse repérée par l'étiquette fun. Dans le corps de la fonction, la valeur de la $i$ème variable est disponible à l'adresse (`BP` + $2(n - i + 1)$). Au retour, le bloc d'activation est oté de la pile et `BP` est restauré à sa valeur initiale. Le résultat de l'appel fonctionnel est disponible dans `AX`. Une optimisation évite des sauvegardes inutiles lorsque qu'une fonction n'a pas d'arguments.

La spécification de la figure 2.2 emploie un environnement, une fonction E : Var $\rightarrow$ $\mathbb{N}$, afin de mémoriser l'adresse relative (offset) dans la pile de l'argument associé à une variable. Cet environnement n'intervient (et n'a de sens) qu'au moment de la compilation.

Soit par exemple la déclaration dec_{nand} suivante :

```
fun nand(X,Y) = if X then (if Y then false else true) else true
```

Elle est traduite par ***comp*** en :

```
nand : MOV BP,SP
       MOV AX,(BP+4);  CMP AX,0;  JZ lab3
       MOV AX,(BP+2);  CMP AX,0;  JZ lab1
       MOV AX,0;  JMP lab2
lab1 : MOV AX,1
lab2 : JMP lab4
lab3 : MOV AX,1
lab4 : RET 4
```

2.2.3 Machine d'exécution.

La sémantique de cette machine est donnée par une fonction L_{Mach} : Code $\times$ Mem $\rightarrow$ Mem où Code est l'ensemble des codes de programmes pour Mach (une affectation d'instructions aux différentes adresses mémoire de la zone code) et Mem l'ensemble des états possibles des registres, des indicateurs et de la mémoire réservée à la pile.

Soient $T_{\mathcal{P}}$: Dec $\rightarrow$ Code et $T_{\mathcal{D}}$: Fun $\times$ BoolVal* $\rightarrow$ Mem les fonctions qui à dec et $(fun, (v_1, \ldots, v_n))$ associent respectivement $code$ et mem vérifiant :

- $code$ est une affectation en mémoire de $\boldsymbol{comp}[\![dec]\!]$,
- mem est une mémoire telle que :

- les valeurs $v_1, \ldots, v_n$, *hlt* sont rangées par ordre décroissant à une adresse $addr$ de la zone données,
- $\texttt{SP} = addr - 2(n+1)$,
- `IP` est positionné à l'adresse repérée par « *fun* : ».

Soit $T_{\mathcal{R}}$: Mem $\rightarrow$ BoolVal la fonction qui à un état mem de la mémoire et des registres dans lequel `AX` vaut 1 (resp. 0) associe le booléen $true$ (resp. *false*). Alors $(T_{\mathcal{P}}, T_{\mathcal{D}}, T_{\mathcal{R}})$ est une *compilation* (déf. 2.3) de $Bool$ dans L_{Mach}.

Dans le cas de $\boldsymbol{comp}[\![dec_{\texttt{nand}}]\!]$, lorsqu'on lance l'exécution à l'adresse repérée par l'étiquette `nand` sur une pile contenant successivement `1`, `0`, *hlt*, la machine s'interrompt après l'exécution de onze instructions avec la valeur `1` dans le registre `AX`. Autrement dit, $Bool(\texttt{nand}\,(true, \mathit{false})) = true$.

2.2.4 Modèle d'exécution.

Soient Instr^* l'ensemble des séquences d'*instructions* de Mach : $\mathrm{Code} \times \mathrm{Mem} \rightarrow \mathrm{Instr}^*$, et M_{Mach} la fonction qui à $(code, mem) \in \mathrm{Code} \times \mathrm{Mem}$ associe la séquence des instructions effectués lors de l'exécution de $code(mem)$. M_{Mach} définit un *modèle d'exécution* de L_{Mach}. On en déduit un modèle d'exécution de BOOL compilé (déf. 2.4) par l'intermédiaire de la fonction M_{Comp} : $\mathrm{Dec} \times (\mathrm{Fun} \times \mathrm{BoolVal}^*) \rightarrow \mathrm{Instr}^*$ définie par $M_{\mathrm{Comp}} = M_{\mathrm{Mach}} \circ (T_{\mathcal{P}}, T_{\mathcal{D}})$.

La *trace d'exécution* de $\texttt{nand}\,(true, \mathit{false})$ dans M_{Comp}, qui est aussi celle dans M_{Mach} de $\boldsymbol{comp}[\![dec_{\texttt{nand}}]\!]$ sur une pile qui contient les valeurs successives `1` et `0`, est la séquence :

$$\langle\!\langle \texttt{MOV BP,SP; MOV AX,(BP+4); CMP AX,0; JZ } lab_3\texttt{; MOV AX,(BP+2); CMP AX,0; JZ } lab_1\texttt{;}$$
$$\texttt{MOV AX,1; JMP } lab_4\texttt{; RET 4}\rangle\!\rangle$$

2.3 Modèles d'exécution pratiques.

Si l'on modélise l'exécution, c'est avec le but de formuler la *performance* des programmes (chap. 3). On veut pour cela rester *fidèle* au comportement dynamique effectif. Néanmoins, pour des raisons de simplicité, un modèle d'exécution peut *abstraire* certaines informations de la trace.

2.3.1 Fidélité.

Machine concrète.

Le type de trace que nous avons donné à la section §2.2.4 n'est pas une vision parfaite de l'exécution. Bien que le modèle soit complet, il ne caractérise pas totalement le comportement dynamique de la machine :

- Le nombre de cycles nécessaires pour exécuter une instruction est une mesure de la performance d'une machine physique. Celui-ci peut varier en fonction de la position de l'instruction en mémoire. C'est le cas par exemple pour l'INTEL 8086, qui nécessite quelques cycles supplémentaires lorsque l'accès à un mot se fait à partir d'une adresse impaire [DG82].

- Dans la trace ne figure pas non plus les effets de cache, de synchronisation, de gestion mémoire (MMU), de mémoire virtuelle, de file d'attente (pipe-line) des instructions du processeur... Au mieux, on ne dispose que d'*encadrements* des temps d'exécution.

- Les *instructions élémentaires* qui constituent la trace ne spécifient pas *explicitement* tout le comportement dynamique, et il faut parfois rejouer l'exécution pour connaître les opérations réellement effectuées. Cela vient du fait que certaines opérations dépendent de la valeur des indicateurs, des registres ou de la mémoire, valeur qui n'apparaît pas dans le texte seul de l'instruction. En voici des exemples :

  – Dans la trace d'exécution donnée à la section §2.2.4 le premier « `JZ` » n'a pas provoqué de branchement car `ZF` n'était pas positionné, alors que le second, au contraire, a effectivement provoqué un saut à l'adresse lab_1.
  – L'instruction « `REP CMPS` » du 8086 compare deux zones mémoire sur une longueur spécifiée par le registre `CX` ; elle s'interrompt lorsque `CX` s'annule ou lorsque le parcours des deux zones permet de déceler une différence. La quantité de cycles pour exécuter cette instruction est une fonction affine du nombre d'itérations, nombre qui dépend de la valeur du registre `CX` et du contenu de la mémoire.

Machine abstraite.

Ce dernier point (le temps d'exécution d'une instruction dépend de la valeur des registres ou de la mémoire) est un cas encore plus fréquent pour les machines abstraites. Prenons l'exemple de la WAM, machine abstraite de Warren [War83, AK90] :

– L'instruction « `unify_variable X`n » a deux comportements dynamiques distincts suivant la valeur de l'indicateur $mode$ (selon que l'on est en train de lire ou d'écrire un terme). Mais leur temps d'exécution respectif est constant.
– L'instruction « `get_value X`n`,A`i » nécessite l'unification des deux termes pointés par `X`n et `A`i ; elle dépend de la valeur des registres `X`n et `A`i, et du contenu de la mémoire.

De manière générale, pour les instructions des machines concrètes ou abstraites, *la performance dépend du contexte*. Que la trace soit la suite des instructions exécutées suppose en outre que la machine est *séquentielle* : le comportement dynamique est une succession d'opérations élémentaires. Nous n'étudions pas le cas d'une machine *parallèle*, contrète ou abstraite, qui nécessite une trace arborescente.

Compilation.

A cela s'ajoutent, dans le cas de langages de haut niveau, des problèmes liés à l'environnement ou à la compilation vers une machine, physique ou abstraite :

- La traduction que réalise un compilateur est rarement « spécifiée » autrement que par son implémentation. La complexité d'une telle *spécification* est rédhibitoire si le modèle du langage compilé veut prendre en compte tous les choix du compilateur, depuis le mode d'allocation des registres jusqu'à l'optimisation du taux de remplissage de la file d'attente du processeur.

- Certaines ressources ne sont pas totalement du ressort du compilateur mais plutôt du *système d'exploitation* : mode d'allocation et de libération de l'espace mémoire, gestion des pages mémoire selon la localisation des données[3], etc.

- Pour que la spécification soit complète, il est indispensable d'inclure, le cas échéant, une formalisation du glaneur de cellules (garbage collector).

Compromis.

Dans notre modélisation, nécessairement simplificatrice, nous supposons que le comportement dynamique des opérations de la machine d'exécution dépend uniquement des instructions élémentaires et non de l'*histoire de l'exécution*. Pour cela, nous annotons chaque instruction d'une machine concrète ou abstraite par des informations contextuelles qui multiplient le jeu d'instructions. On forme ainsi des *traces d'exécution élémentaires*.

- Si l'on veut prendre en compte l'influence de l'*alignement*, de l'affectation des instructions en mémoire, on annote chaque instruction de la trace par une abstraction de sa position en mémoire. Pour le 8086 par exemple, c'est simplement un bit représentatif de la parité.

- Si l'on veut prendre en compte la file d'attente du processeur, il faut la faire apparaître dans la machine d'exécution. On matérialise à l'aide de pseudo-instructions les opérations atomiques de vidange de queue (lors d'un branchement), d'attente de disponibilité (premier chargement dans la file d'attente) et de chargement effectif (accès mémoire).

- On fait apparaître au côté des instructions les valeurs des indicateurs, registres et mémoire qui influencent leur comportement dynamique. Par exemple, les instructions qui dépendent de la valeur d'un indicateur sont dédoublées pour refléter le contexte dans lequel elles sont effectuées : on distingue ainsi les traces élémentaires « $\mathtt{JZ}_{\mathtt{ZF=0}}\ addr$ » et « $\mathtt{JZ}_{\mathtt{ZF=1}}\ addr$ », ou bien « $\mathtt{unify\_variable}_{mode=\mathtt{read}}\ \mathtt{X}n$ » et « $\mathtt{unify\_variable}_{mode=\mathtt{write}}\ \mathtt{X}n$ ».

  Pour la WAM, on a plus généralement des traces comme « $\mathtt{get\_value}\ \mathtt{X}n(t)\mathtt{,A}i(t')$ » où t et t' sont les termes pointés respectivement par $\mathtt{X}n$ et $\mathtt{A}i$. Plus précisément, t et t' sont des *codages* de ces termes, car un terme peut être représenté de différentes manières suivant l'histoire de sa construction, du fait du chaînage des références.

La trace d'exécution de $\mathtt{nand}\,(true, false)$ selon un modèle qui modifie M_{Mach} (cf. §2.2.4) pour prendre en compte l'état instantané des registres et de la mémoire devient ainsi $\langle\!\langle \ldots;\ \mathtt{JZ}_{\mathtt{ZF=0}}\ lab_3;\ \ldots;\ \mathtt{JZ}_{\mathtt{ZF=1}}\ lab_1;\ \ldots\rangle\!\rangle$.

En ce qui concerne les langages de haut niveau :

- Si l'on n'a pas de modèle de gestion de mémoire, on ne peut prendre en compte les *variations* de performance. Allocation et libération de mémoire ont alors un coût qui ne dépend que de la taille des zones manipulées.

- Sans modèle du glanage de cellules, on ne peut que comptabiliser des ajustements grossiers au moment de l'allocation et à la libération explicite de mémoire. Cette hypothèse simplificatrice donne une image très imprécise de la performance de certains programmes.

[3]Lorsque qu'un programme manipule des données de taille importante qui ne peuvent résider en totalité dans la mémoire physique de l'ordinateur, le temps des accès disque peut devenir prépondérant : nous avons rencontré des programmeurs FORTRAN qui obtenaient des gains *majeurs* dans les zones critiques de gros codes scientifiques en découpant en tranches leurs tableaux afin de les faire cadrer dans les pages mémoire…

2.3.2 Abstraction.

D'un autre côté, la trace d'exécution, telle qu'elle est donnée à la section §2.2.4, contient aussi des informations superflues car le comportement dynamique de beaucoup d'instructions paramétrées est indépendant de la valeur effective des arguments.

Ainsi, au lieu d'enregistrer les traces $\langle\!\langle \texttt{JMP}\ lab_4;\ \ \texttt{MOV AX,0};\ \ \texttt{MOV AX,(BP+2)} \rangle\!\rangle$, on fera figurer des paramètres anonymes : $\langle\!\langle \texttt{JMP}\ addr;\ \ \texttt{MOV}\ reg\texttt{,}imm;\ \ \texttt{MOV}\ reg\texttt{,(}reg + depl\texttt{)} \rangle\!\rangle$. Une telle trace est *abstraite* dans le sens où elle ne fait figurer que les informations caractéristiques du comportement à l'exécution.

Par exemple, la trace d'exécution de $\texttt{nand}\,(true, false)$ selon un modèle $M'_{\mathrm{Mach}} : \mathrm{Code} \times \mathrm{Mem} \rightarrow \mathrm{Instr}'^{*}$, qui n'abstrait que les déplacements et les addresses dans les instructions de M_{Mach} (cf. §2.2.4), est :

$$\langle\!\langle \texttt{MOV BP,SP};\ \texttt{MOV AX,(BP+}depl\texttt{)};\ \texttt{CMP AX,0};\ \texttt{JZ}_{\texttt{ZF=0}}\ addr;\ \texttt{MOV AX,(BP+}depl\texttt{)};\ \texttt{CMP AX,0};\ \texttt{JZ}_{\texttt{ZF=1}}\ addr;\ \texttt{MOV AX,1};\ \texttt{JMP}\ addr;\ \texttt{RET 4} \rangle\!\rangle$$

2.3.3 Une sémantique opérationnelle comme modèle d'exécution.

Totalement spécifier une implémentation est une tâche en soi. On peut en juger par la somme de détails (indispensables) mentionnés à la section §2.2 pour spécifier l'implémentation de BOOL, langage ridiculement simple. Et nous n'avons donné que les points essentiels.

Bien que l'implémentation visée soit en un certain sens le meilleur support pour l'exécution (c'est « par définition » le plus fidèle), on peut lui préférer des modèles moins fidèles, mais qui permettent une spécification plus abstraite.

C'est pourquoi nous examinons dans la suite comment considérer des sémantiques opérationnelles comme des machines d'exécution. Les principaux intérêts d'une telle modélisation sont les suivants.

- Une spécification sémantique permet une analyse de plus haut niveau sans trop s'éloigner pour cela des implémentations réelles.

- Il n'est pas indispensable de spécifier *explicitement* l'implémentation que l'on désire modéliser (que les programmes soient ou non compilés dans un autre langage puis interprétés). Les règles sémantiques sont alors simplement accompagnées d'une justification informelle qui « esquisse » l'implémentation *visée* en s'appuyant sur l'*exécution* de la *spécification sémantique*. Cette justification comporte aussi en pratique une formulation des coûts associés à chaque comportement dynamique (cf. §3.2.5, liens avec une implémentation informelle).

- Des propriétés générales peuvent être étudiées au niveau des *formalismes sémantiques*, et non spécifiquement, pour chaque implémentation.

Nous prenons comme exemples dans les sections suivantes la sémantique naturelle et les schémas de programme récursifs.

D'autres auteurs ont eu une approche similaire. Le schéma général de complexité proposé par Gurr [Gur91] est également fondé sur des formalismes sémantiques : la sémantique dénotationnelle et une formulation catégorique de la sémantique et de l'exécution proposée par Moggi [Mog89b, Mog90]. Sands emploie une spécification opérationnelle en sémantique naturelle pour étudier la performance dans un langage fonctionnel paresseux [San90, San91, San93].

Hannan et Miller ont des préoccupations voisines [HM90, Han91] ; ils étudient comment transformer des spécifications opérationnelles de sémantique naturelle en divers types de machines abstraites. Beaucoup d'autres modélisations de l'exécution sont en fait indirectes : elles étudient la complexité dans un langage particulier en employant, implicitement ou non, une sémantique opérationelle.

2.4 Sémantique naturelle.

La *sémantique naturelle* [Kah87] (baptisée à l'origine *sémantique opérationnelle structurelle* [Plo81]) est un bon support de modèles d'exécution parce qu'elle est à la fois simple et souple ; elle formalise aussi bien les détails de bas niveau d'une implémentation que des spécifications plus déclaratives.

Une spécification en sémantique naturelle est un ensemble de règles divisées en deux parties : au numérateur se trouvent les *hypothèses*, ensemble de *jugements* (ou *séquents*) et de *formules logiques* (qui déterminent les conditions d'application de la règle) ; le dénominateur est la *conclusion*, formée d'un unique jugement. Des variables permettent d'instancier les règles et de les composer (le jugement d'un dénominateur est identifié avec un jugement d'un numérateur) pour construire des *arbres de preuve*. Le plus souvent, un *jugement* de sémantique naturelle est de la forme « $A \vdash sujet \Rightarrow B$ », où A et B peuvent être des listes d'objets ; il doit être compris comme « dans le contexte A, le terme $sujet$ se traduit par B ».

Nos notations sont également inspirées de la définition formelle de STANDARD ML [MTH90, MT91] :

- Si E et E' sont des fonctions de $X \to Y$, on note $E + E'$ la fonction de domaine $Dom(E + E') = Dom(E) \cup Dom(E')$ telle que $(E + E')(x) =$ si $x \in Dom(E')$ alors $E'(x)$ sinon $E(x)$ pour tout $x \in X$. Cette opération est employée pour combiner des environnements.
- Si E_1, E'_1 et E_2, E'_2 sont des fonctions de $X_1 \to Y_1$ et $X_2 \to Y_2$, et si $E = (E_1, E_2)$, on note :
  - $E + E'_1 = (E_1 + E'_1, E_2)$ et $E + E'_2 = (E_1, E_2 + E'_2)$
  - $E(x_1) = E_1(x_1)$ et $E(x_2) = E_2(x_2)$ pour $x_1 \in X_1,\ x_2 \in X_2$

  Cette convention simplifie la manipulation groupée d'environnements de différente nature.

Nous prenons BOOL comme support de la présentation.

2.4.1 Valeurs.

L'ensemble des *objets sémantiques* employés dans la sémantique naturelle de BOOL est donné à la figure 2.3, accompagné de quelques exemples.

Bien que « Exp » soit une classe (un ensemble) *syntaxique*, il apparaît aussi dans les objets *sémantiques* pour le codage des définitions de fonctions.

Nous ne précisons pas de caractéristiques dynamiques pour les environnements ; nous les considérons comme des fonctions qui associent des booléens aux noms de variable. L'environnement FE, qui associe un « code abstrait » aux noms des fonctions BOOL, n'intervient pas matériellement dans l'évaluation ; il joue le rôle d'une compilation. En vertu des notations ci-dessus, un environnement $E = (FE, VE)$ permet d'accéder à la définition d'une fonction par $E(fun) = FE(fun)$ et à la valeur d'une variable par $E(var) = VE(var)$.

var	$\in$	Var	$X,\ Y,\ Z\ldots$
fun	$\in$	Fun	$or,\ nand\ldots$
v	$\in$	BoolVal	$true,\ false$
VE	$\in$	VarEnv = Var $\rightarrow$ BoolVal	$\{X \mapsto true, Y \mapsto false\}$
FE	$\in$	FunEnv = Fun $\rightarrow$ Var* $\times$ Exp	$\{not \mapsto (X, \texttt{if X then false else true})\}$
E	$\in$	FunEnv $\times$ VarEnv	$(\{identity \mapsto (Z, \texttt{not(not(Z))})\}, \{X \mapsto true\})$

Figure 2.3 : Objets sémantiques de BOOL

2.4.2 Sémantique.

La figure 2.4 spécifie en termes de jugements sémantiques comment évaluer $fun(v_1, \ldots, v_n)$ dans le contexte d'une déclaration dec. La première partie de cette preuve, celle du jugement « $dec \Rightarrow FE$ » (fig. 2.5), n'est pas strictement indispensable à l'évaluation ; elle facilite simplement l'accès aux définitions de fonctions en les rangeant dans un environnement. C'est la seconde partie, la preuve du jugement « $E \vdash exp \Rightarrow v$ » (fig. 2.6), qui réalise l'évaluation proprement dite[4].

Cette définition a bien un sens car, puisque tout jugement ne peut avoir qu'une seule preuve, il n'y a au plus qu'un résultat v. En effet, les sujets des règles sont mutuellement exclusifs, excepté pour la construction `if` dont le déterminisme est garanti grâce à des hypothèses contradictoires, et les axiomes ne comportent pas de variables libres car les environnements (E, FE, VE) et les sujets (dec, exp, fun, var) sont toujours instanciés. Notons qu'il n'y a pas d'erreur de typage possible dans BOOL. La seule raison qu'a un programme de ne pas être défini sur l'une de ses données est la non-terminaison. Il faut pour cela un appel fonctionnel récursif, car il n'y a pas d'autre construction itérative.

La fonction « `fun nand(X,Y) = if X then (if Y then false else true) else true` » définie précédemment (cf. §2.2.2) vérifie bien $Bool(\texttt{nand}\,(true, false)) = true$ au moyen des règles de sémantique naturelle.

2.4.3 Machine d'exécution.

Une spécification en sémantique naturelle S peut être considérée comme une *machine d'exécution abstraite* : chaque règle sémantique $\rho \in S$ correspond à une opération élémentaire de cette machine, et exécuter un programme (l'arbre de syntaxe abstraite) consiste à construire la preuve (l'arbre de preuve) du jugement correspondant. Cela nécessite une spécification *déterministe* [Att89] (aussi appelée *non-ambiguë* [AC90]) : tout jugement a au plus *une* preuve, c'est-à-dire *une* exécution.

Il existe d'ailleurs des implémentations concrètes de la sémantique naturelle, au moyen de PROLOG ou de grammaires attribuées. Les spécifications sont décrites dans le formalisme TYPOL, puis traduites et exécutées [Des83, Des84, Des88]. Dans l'implémentation en PROLOG, chaque règle est traduite et interprétée comme une clause logique ; l'exécution est la preuve par réfutation de PROLOG, qui progresse essentiellement de la racine vers les feuilles, et de gauche à droite (pour les prémisses). L'implémentation en termes d'attributs sémantiques

[4] Il semble y avoir redondance entre la définition de la figure 2.4 et la règle ρ_{call_n} de la figure 2.6 : il eut été simple de spécifier la sémantique de BOOL par « $Bool(dec, exp) = v$ ssi $dec \Rightarrow FE$ et $FE, \varnothing \vdash exp \Rightarrow v$ ». Cependant on perd dans ce cas la distinction entre programme et donnée, qui est fondamentale pour la mesure de performance (chap. 3).

$$Bool(dec, (fun, (v_1, \ldots, v_n))) = v$$
$$\text{ssi}$$
$$dec \Rightarrow FE \;\text{ et }\; FE(fun) = (var_1, \ldots, var_n, exp) \;\text{ et }\; FE, \{var_1 \mapsto v_1, \ldots, var_n \mapsto v_n\} \vdash exp \Rightarrow v$$

Figure 2.4 : Sémantique naturelle de BOOL

$$\frac{}{fun\texttt{(}var_1\texttt{,} \ldots \texttt{,}var_n\texttt{)} \;\texttt{=}\; exp \Rightarrow \{fun \mapsto (var_1, \ldots, var_n, exp)\}}$$

$$\frac{funbind \Rightarrow FE \qquad \langle funbinds \Rightarrow FE' \rangle}{funbind \;\langle \texttt{and}\; funbinds \rangle \Rightarrow FE \;\langle + FE' \rangle}$$

$$\frac{funbinds \Rightarrow FE}{\texttt{fun}\; funbinds \Rightarrow FE}$$

Figure 2.5 : Sémantique naturelle des déclarations de BOOL

$$\frac{}{E \vdash \texttt{true} \Rightarrow true} \qquad (\rho_{\mathsf{true}})$$

$$\frac{}{E \vdash \texttt{false} \Rightarrow false} \qquad (\rho_{\mathsf{false}})$$

$$\frac{}{E \vdash var \Rightarrow E(var)} \qquad (\rho_{\mathsf{var}})$$

$$\frac{E \vdash exp_1 \Rightarrow true \qquad E \vdash exp_2 \Rightarrow v}{E \vdash \texttt{if}\; exp_1 \;\texttt{then}\; exp_2 \;\texttt{else}\; exp_3 \Rightarrow v} \qquad (\rho_{\mathsf{if\ true}})$$

$$\frac{E \vdash exp_1 \Rightarrow false \qquad E \vdash exp_3 \Rightarrow v}{E \vdash \texttt{if}\; exp_1 \;\texttt{then}\; exp_2 \;\texttt{else}\; exp_3 \Rightarrow v} \qquad (\rho_{\mathsf{if\ false}})$$

$$\frac{\begin{array}{c} E \vdash exp_1 \Rightarrow v_1 \quad \ldots \quad E \vdash exp_n \Rightarrow v_n \\ E(fun) = (var_1, \ldots, var_n, exp) \qquad E + \{var_1 \mapsto v_1, \ldots, var_n \mapsto v_n\} \vdash exp \Rightarrow v \end{array}}{E \vdash fun\texttt{(}exp_1\texttt{,} \ldots \texttt{,}exp_n\texttt{)} \Rightarrow v} \qquad (\rho_{\mathsf{call}_n})$$

Figure 2.6 : Sémantique naturelle des expressions de BOOL

[Att89, AC90] fait une traduction fonctionnelle de certaines classes de programme TYPOL déterministes. Ces implémentations font partie du système CENTAUR [BCD+88, JR92].

Si la spécification en sémantique naturelle est seul support de la sémantique et de l'exécution (cf. §2.3.3), cela suppose que les règles sémantiques sont de nature opérationnelle et en particulier qu'à chaque étape de la construction d'une preuve, une seule règle est applicable à l'arbre de preuve courant : c'est la condition de *pseudo-déterminisme* [Att89]. Lorsqu'il y a ambiguïté, comme c'est parfois le cas des constructions conditionnelles, il faut fournir une formulation opérationnelle équivalente.

Ainsi les règles de la figure 2.6 décrivent-elles une machine d'exécution pour BOOL. Afin qu'elles « suivent » de manière plus apparente l'implémentation de BOOL dans la machine Mach (cf. §2.2), nous donnons une formulation opératoire équivalente pour désambiguïser le choix des règles non-pseudo-deterministes $\{\rho_{\textsf{if true}}, \rho_{\textsf{if false}}\}$ lors d'une construction de l'arbre de preuve depuis la racine vers les feuilles ; c'est l'objet des nouvelles règles de la figure 2.7.

$$\frac{E \vdash exp_1 \Rightarrow v \qquad E, v \vdash exp_2, exp_3 \Rightarrow v'}{E \vdash \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \Rightarrow v'} \qquad (\rho_{\textsf{if}})$$

$$\frac{E \vdash exp_2 \Rightarrow v}{E, true \vdash exp_2, exp_3 \Rightarrow v} \qquad (\rho_{\textsf{case true}})$$

$$\frac{E \vdash exp_3 \Rightarrow v}{E, false \vdash exp_2, exp_3 \Rightarrow v} \qquad (\rho_{\textsf{case false}})$$

Figure 2.7 : Règles opératoires de `if`

2.4.4 Modèle d'exécution.

Nous prenons comme *trace d'exécution* d'un jugement une *observation* de son arbre de preuve. À la différence de la trace d'exécution pour un machine physique ou abstraite, qui est une liste linéaire d'instructions (cf. §2.2.4), la trace d'exécution est ici un objet arborescent ; cela reflète à la fois l'indépendance de l'ordre des prémisses dans la preuve d'un jugement, et le fait que la sémantique naturelle permet aussi de modéliser des exécutions parallèles.

De la même manière que certaines instructions de machines concrètes ou abstraites dépendent du contexte d'exécution (indicateurs, registres, mémoire, cf. §2.3.1), le comportement dynamique que l'on associe à une règle peut dépendre de la valeur des variables qui y figurent.

Aussi le codage général des traces d'exécution est du type de celui employé pour coder les arbres de preuve dans les formalismes logiques comme LF [HHP91] : à chaque règle sémantique $\rho \in S$ est associée une constante (aussi notée ρ) de type fonctionnel dont les arguments sont d'une part les valeurs des variables $x_1, \ldots, x_{n_\rho}$ qui apparaissent dans la règle et d'autre part les preuves $\pi_1, \ldots, \pi_{n'_\rho}$ des prémisses ; l'*objet preuve* est un terme construit à l'aide de ces opérateurs[5].

[5] Une autre formulation des constructeurs d'arbres de preuve incorpore à ces opérateurs la valeur des variables qui figurent sur les régles. Cela revient à considérer la famille d'opérateurs $(\rho_{v_1,\ldots,v_{n_\rho}})_{\rho \in S, v_1 \in V_1^\rho, \ldots, v_{n_\rho} \in V_{n_\rho}^\rho}$ constituée des applications partielles des opérateurs ρ sur les valeurs $v_i \in V_i^\rho$, où l'ensemble V_i^ρ est le domaine de variation de la variable x_i de ρ ; les nouvelles constantes $\rho_{v_1,\ldots v_{n_\rho}}$ sont des opérateurs qui ont pour arité le nombre de prémisses de ρ. C'est la formulation adoptée pour les termes représentant

En pratique, les variables qui représentent des sous-termes du *sujet* (voir [Kah87, Att89]) d'un jugement ne sont pas comptabilisées car elles sont toujours instanciées par un arbre de syntaxe abstraite qui représente un sous-terme du programme.

Considérons ainsi $M_{\mathrm{S.Nat}}$, modèle d'exécution de BOOL en sémantique naturelle, avec les régles pseudo-déterministes pour `if`. Une trace d'exécution dans $M_{\mathrm{S.Nat}}$ est un terme de l'algèbre libre construite sur la signature F constituée des opérateurs suivants :

- $\rho_{\mathsf{true}},\ \rho_{\mathsf{false}}$: $\mathrm{env} \rightarrow \mathrm{proof}$
- ρ_{var} : $\mathrm{env} \times \mathrm{var} \times \mathrm{bool} \rightarrow \mathrm{proof}$
- ρ_{if} : $\mathrm{env} \times (\mathrm{proof} \times \mathrm{bool})^2 \rightarrow \mathrm{proof}$
- $\rho_{\mathsf{case\,true}},\ \rho_{\mathsf{case\,false}}$: $\mathrm{env} \times (\mathrm{proof} \times \mathrm{bool}) \rightarrow \mathrm{proof}$
- ρ_{call_n} : $\mathrm{env} \times (\mathrm{proof} \times \mathrm{bool})^n \times \mathrm{fun} \times \mathrm{var}^n \times (\mathrm{proof} \times \mathrm{bool}) \rightarrow \mathrm{proof}$ pour $n \in \mathbb{N}$

Dans le cas de BOOL, nous ne modélisons que l'évaluation proprement dite des expressions, c'est-à-dire la preuve du jugement « $E \vdash exp \Rightarrow v$ ». Celle de « $dec \Rightarrow FE$ » fait office de compilation et n'a pas d'influence sur le comportement dynamique ; elle n'apparaît pas dans la trace.

Par exemple, l'exécution du programme $\mathtt{nand}\,(true, \mathit{false})$ doit construire la preuve du jugement « $E \vdash$ `if X then (if Y then false else true) else true` $\Rightarrow true$ » avec $E = \{X \mapsto true, Y \mapsto \mathit{false}\}$. Sa trace par $M_{\mathrm{S.Nat}}$ est :

$$
\begin{aligned}
&M_{\mathrm{S.Nat}}(\mathtt{nand}\,(true, \mathit{false})) = \\
&\quad \rho_{\mathsf{if}}(E, \rho_{\mathsf{var}}(E, X, true), true, \rho_{\mathsf{case\,true}}(E, \\
&\qquad \rho_{\mathsf{if}}(E, \rho_{\mathsf{var}}(E, Y, \mathit{false}), \mathit{false}, \rho_{\mathsf{case\,false}}(E, \rho_{\mathsf{true}}(E), true), true), true), true)
\end{aligned}
$$

En vertu du principe d'abstraction (cf. §2.3.2), on peut ne faire figurer dans une trace que les éléments qui ont une influence sur le comportement dynamique[6]. Soit par exemple $M'_{\mathrm{S.Nat}}$ un modèle d'exécution abstrait de BOOL en sémantique naturelle, qui élimine la dépendance dans les variables et dans l'environnement. Une trace d'exécution dans $M'_{\mathrm{S.Nat}}$ est alors un terme de l'algèbre libre construite sur la signature F' suivante :

- $F' = \{\rho_{\mathsf{true}} : 0,\ \rho_{\mathsf{false}} : 0,\ \rho_{\mathsf{var}} : 0,\ \rho_{\mathsf{if}} : 2,\ \rho_{\mathsf{case\,true}} : 1,\ \rho_{\mathsf{case\,false}} : 1,\ \rho_{\mathsf{call}_n} : n+1 \mid n \in \mathbb{N}\}$

La trace d'exécution de $\mathtt{nand}\,(true, \mathit{false})$ devient :

$$
M'_{\mathrm{S.Nat}}(\mathtt{nand}\,(true, \mathit{false})) = \rho_{\mathsf{if}}(\rho_{\mathsf{var}}, \rho_{\mathsf{case\,true}}(\rho_{\mathsf{if}}(\rho_{\mathsf{var}}, \rho_{\mathsf{case\,false}}(\rho_{\mathsf{true}}))))
$$

Ainsi, le comportement dynamique associé à la règle ρ_{var} est indépendant de l'environnement E et de la variable var, c'est-à-dire que l'on considère que l'accès à la valeur d'une variable a un coût constant. Lorsque le comportement n'est pas uniforme, il faut spécifier les caractéristiques dynamiques des opérations sur l'environnement : stockage « $E + \{var \mapsto v\}$ » et accès « $E(var)$ ».

La spécification de BOOL au moyen des règles non-pseudo-déterministes pour `if` conduit à des modèles d'exécution similaires à ceux qui emploient les règles pseudo-déterministes. Ainsi, l'observation $M''_{\mathrm{S.Nat}}$ déduite des règles $\{\rho_{\mathsf{if\,true}}, \rho_{\mathsf{if\,false}}\}$ emploie une signature F'' définie par :

- $F'' = \{\rho_{\mathsf{true}} : 0,\ \rho_{\mathsf{false}} : 0,\ \rho_{\mathsf{var}} : 0,\ \rho_{\mathsf{if\,true}} : 2,\ \rho_{\mathsf{if\,false}} : 2,\ \rho_{\mathsf{call}_n} : n+1 \mid n \in \mathbb{N}\}$

---

les preuves dans [Hue88].

[6]Cela revient à identifier certains opérateurs de la famille $(\rho_{v_1,\dots v_{n_\rho}})_{v_1 \in V_1^\rho, \dots, v_{n_\rho} \in V_{n_\rho}^\rho}$ lorsque le comportement dynamique associé à une règle ρ est indépendant de la valeur de certains v_i.

Dans le cas de `nand`, on a par exemple :

$$M''_{\mathrm{S.Nat}}(\mathtt{nand}\,(true, false)) = \rho_{\mathsf{if\ true}}(\rho_{\mathsf{var}}, \rho_{\mathsf{if\ false}}(\rho_{\mathsf{var}}, \rho_{\mathsf{true}}))$$

Après la spécification de modèles d'exécution au moyen de la sémantique naturelle, nous examinons à présent comment faire de même à l'aide des schémas de programme récursifs.

2.5 Schémas de programme.

Le formalisme des *schémas de programme* est un outil d'étude des langages de programmation [Cou90a]. Il permet de spécifier la sémantique d'un langage, d'établir des relations entre classes de programmes, d'étudier les propriétés de terminaison et de valider des transformations. Son champ d'application dépasse en fait le domaine de la programmation pure et s'étend à la plupart des constructions qui font apparaître des définitions récursives, comme par exemple les grammaires.

Les schémas de programme récursifs bénéficient de la généralité de l'approche algébrique. Bien que tombés en désuétude, ils restent un bon support général des problèmes d'évaluation partielle du fait de la variété de leur sémantique opérationnelle, et parce qu'ils explicitent dans une certaine mesure les mécanismes d'appels de procédure et permettent ainsi de traiter de manière générique les transformations de programme. Les transformations majeures de l'évaluation partielle sont d'ailleurs dérivées des transformations fold/unfold proposées initialement par Burstall et Darlington sur des schémas récursifs [BD76, BD77].

2.5.1 Aperçu.

Syntaxe. Un *schéma de programme* est un terme, le plus souvent infini, décrit par un *système d'équations récursives*. Par exemple,

$$\mathbf{fac}(n) \quad = \quad \mathsf{ifz}(n, 1, \mathsf{mul}(n, \mathbf{fac}(\mathsf{sub1}(n))))$$

Cette équation représente le terme infini suivant.

$$\begin{array}{l}\mathsf{ifz}(n, 1, \mathsf{mul}(n,\\ \quad \mathsf{ifz}(\mathsf{sub1}(n), 1, \mathsf{mul}(\mathsf{sub1}(n),\\ \quad\quad \mathsf{ifz}(\mathsf{sub1}(\mathsf{sub1}(n)), 1, \mathsf{mul}(\mathsf{sub1}(\mathsf{sub1}(n)), \ldots))))))\end{array}$$

Notons que le symbole **fac** n'apparaît plus. Son occurrence dans l'équation récursive a pour seul but de décrire la construction du terme infini obtenu par *dépliages* successifs de l'équation **fac**.

Interprétation. Pour donner un sens à ce schéma, il faut dire comment interpréter les symboles qui y figurent. Si l'on prend comme domaine les nombres entiers et que l'on donne comme signification aux symboles ifz, 1, sub1 et mul respectivement le test d'égalité à zéro (noté *ifz*), l'entier 1, la soustraction par 1 et la multiplication, le schéma **fac** représente bien alors la fonction factorielle ordinaire[7].

[7] Il faut noter la séparation entre *schéma* et *interprétation* : le même schéma **fac** représente une fonction `reverse` de retournement des listes si l'on prend comme domaine les listes et comme interprétation des symboles ifz, 1, sub1 et mul respectivement le test d'égalité à la liste vide, la liste vide, la concaténation des listes privées de leur élément de tête et l'extraction de la tête de liste. Réciproquement, plusieurs schémas différents peuvent représenter une même fonction.

Sémantique. Il y a principalement deux moyens de spécifier la *sémantique* d'un schéma de programme, l'une, *opérationnelle*, à l'aide d'un système de réécriture, l'autre, plus déclarative, à l'aide d'un *plus petit point fixe*. L'évaluation du schéma de programme $\mathbf{fac}(3)$ suivant une sémantique opérationnelle consiste en la dérivation suivante (les expressions à réduire sont soulignées) :

$$\begin{array}{rcl}
\underline{\mathbf{fac}}(3) & \to & \underline{\mathsf{ifz}}(3,1,\mathsf{mul}(3,\mathbf{fac}(\mathsf{sub1}(3)))) \\
& \to & \mathsf{mul}(3,\mathbf{fac}(\underline{\mathsf{sub1}}(3))) \\
& \to & \mathsf{mul}(3,\underline{\mathbf{fac}}(2)) \\
& \xrightarrow{*} & \mathsf{mul}(3,\mathsf{mul}(2,\underline{\mathbf{fac}}(1))) \\
& \xrightarrow{*} & \mathsf{mul}(3,\mathsf{mul}(2,\mathsf{mul}(1,\underline{\mathbf{fac}}(1)))) \\
& \xrightarrow{*} & \underline{\mathsf{mul}}(3,\underline{\mathsf{mul}}(2,\underline{\mathsf{mul}}(1,\underline{1}))) \\
& \xrightarrow{*} & 6.
\end{array}$$

La sémantique en terme de plus petit point fixe fait apparaître les itérations suivantes, où la notation $\perp$ désigne une valeur indéfinie :

$$\begin{array}{lcl}
f^{0}_{\mathbf{fac}}(3) & = & \perp \\
f^{1}_{\mathbf{fac}}(3) & = & \mathit{ifz}(3,1,3.\perp) = \perp \\
f^{2}_{\mathbf{fac}}(3) & = & \mathit{ifz}(3,1,3.\mathit{ifz}(3-1,1,(3-1).\perp)) = \perp \\
f^{3}_{\mathbf{fac}}(3) & = & \mathit{ifz}(3,1,3.\mathit{ifz}(3-1,1,(3-1).\mathit{ifz}(3-1-1,1,(3-1-1).\perp))) = \perp \\
f^{4}_{\mathbf{fac}}(3) & = & \mathit{ifz}(3,1,3.\mathit{ifz}(3-1,1,(3-1).\mathit{ifz}(3-1-1,1,(3-1-1).\mathit{ifz}(3-1-1-1,1,(3-1-1-1).\perp)))) = 6 \\
f^{m}_{\mathbf{fac}}(3) & = & \ldots \; = \; 6 \quad (\text{pour } m \geqslant 5)
\end{array}$$

Moins pratique d'emploi pour simplement déterminer la valeur d'un terme, la sémantique du plus petit point fixe est principalement un outil pour étudier les propriétés des programmes, et en particulier l'équivalence et la correction des transformations.

2.5.2 Définition et sémantiques des schémas de programme.

Les définitions et propositions qui suivent sont en majeure partie tirés d'une synthèse des *schémas de programme récursifs* de Bruno Courcelle [Cou90a], qui fournit également les références bibliographiques essentielles. Elles sont illustrées à la section §2.5.3 sur le langage BOOL. Le lecteur peu familier avec ces définitions aura intérêt à s'y référer. L'annexe N donne les notations et définitions algébriques usuelles, que nous ne rappelons pas ici.

Schéma de programme. Soient $\mathcal{S}$ un ensemble de sortes, F une $\mathcal{S}$-signature, X une $\mathcal{S}$-signature de variables (d'arité nulle), et $\mathbf{\Phi}$ une $\mathcal{S}$-signature d'*inconnues*, disjointe de F.

- Soit $\Phi = \{\varphi_1, \ldots, \varphi_N\} \subset \mathbf{\Phi}$ un ensemble d'inconnues (les φ_i sont distincts deux à deux). Un *système d'équations* sur F avec l'ensemble d'inconnues Φ est un ensemble fini d'équations $\Sigma = \{\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) = t_i \mid i \in [N]\}$ tel que pour tout $i \in [N]$, $X_i = \{x_{i,1}, \ldots, x_{i,n_i}\}$ soit inclus dans X, et tel que les termes t_i et $\varphi_i(x_{i,1}, \ldots, x_{i,n_i})$ soient dans $M(F \cup \Phi, X_i)_{\sigma(\varphi_i)}$. Il existe une et une seule équation par inconnue. On note $Unk(\Sigma) = \Phi$.
- Un système d'équations est *régulier* ssi tous les $\varphi_i \in \Phi$ sont d'arité nulle. Autrement dit, $\Sigma = \{\varphi_i = t_i \mid i \in [N]\}$ avec $t_i \in M(F \cup \Phi)_{\sigma(\varphi_i)}$. Dans le cas contraire, le système est dit *algébrique*.

- Un système d'équations Σ est aussi appelé *schéma de programme récursif*[8] ; on note $\mathbf{\Phi}$ l'ensemble des inconnues et $Sch(F)$ l'ensemble des schémas de programme récursifs sur F et $\mathbf{\Phi}$.

Nous omettons parfois l'indication des sortes afin d'alléger les notations.

Interprétation.

- Une F-*interprétation* est une F-algèbre continue $\boldsymbol{D} = \langle (D_s)_{s\in\mathcal{S}}, (\leqslant_s)_{s\in\mathcal{S}}, (\perp_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{D}})_{f\in F}\rangle$.
- Le domaine $D = (D_s)_{s\in\mathcal{S}}$ peut être considéré comme la signature $\{d : s \mid s \in \mathcal{S},\ d \in D_s\}$, que l'on note également D.
- On peut étendre une F-interprétation $\boldsymbol{D}$ en une $(F \cup D)$-interprétation, aussi notée $\boldsymbol{D}$, par :
  - $\forall d \in D \quad d_{\boldsymbol{D}} = d$
- Pour tout $X_n = (x_1, \ldots, x_n) \in X_{s_1} \times \cdots \times X_{s_n}$ et tout $(d_1, \ldots, d_n) \in D_{s_1} \times \cdots \times D_{s_n}$, on peut étendre une F-interprétation $\boldsymbol{D}$ en une $(F \cup X_n)$-interprétation, notée $\boldsymbol{D}[d_1, \ldots, d_n]$ et constituée de :
  - $\forall f \in F \quad f_{\boldsymbol{D}[d_1,\ldots,d_n]} = f_{\boldsymbol{D}}$
  - $\forall i \in [n] \quad (x_i)_{\boldsymbol{D}[d_1,\ldots,d_n]} = d_i$
- Pour tout $t \in M(F, X)_s$ et $X_n = (x_1, \ldots, x_n) \in X^n$ tel que $Var(t) \subset X_n$, on note $t_{X_n,\boldsymbol{D}}$ la fonction de $D_{\sigma(x_1)} \times \cdots \times D_{\sigma(x_n)} \to D_s$ définie par $t_{X_n,\boldsymbol{D}}(d_1, \ldots, d_n) = (t[x_1/d_1, \ldots, x_n/d_n])_{\boldsymbol{D}} = t_{\boldsymbol{D}[d_1,\ldots,d_n]}$.

Solution d'un schéma de programme.

- Une F-*spécification* $(\Sigma, \boldsymbol{D})$ est un couple constitué d'un F-schéma Σ et d'une F-interprétation $\boldsymbol{D}$.
- Pour tout $i \in [N]$, on note $D_i = D_{\sigma(x_{i,1})} \times \cdots \times D_{\sigma(x_{i,n_i})}$, $\Delta_i = D_i \to D_{\sigma(\varphi_i)}$, et $\widetilde{\Delta} = \Delta_1 \times \cdots \times \Delta_N$. Un n-uplet $(g_1, \ldots, g_N) \in \widetilde{\Delta}$ est une *solution de* Σ *dans* $\boldsymbol{D}$ ssi $g_i(d_1, \ldots, d_{n_i}) = (t_i)_{\boldsymbol{D}[d_1,\ldots,d_{n_i}][g_1,\ldots,g_N]}$ pour tout $i \in [N]$ et tout $(d_1, \ldots, d_{n_i}) \in\ \in D_i$, où $\boldsymbol{D}[d_1, \ldots, d_{n_i}][g_1, \ldots, g_N]$ est la $(F \cup X_i \cup \Phi)$-interprétation constituée de :
  - $\forall f \in F \quad f_{\boldsymbol{D}[d_1,\ldots,d_{n_i}][g_1,\ldots,g_N]} = f_{\boldsymbol{D}}$
  - $\forall j \in [n_i] \quad (x_{i,j})_{\boldsymbol{D}[d_1,\ldots,d_{n_i}][g_1,\ldots,g_N]} = d_j$
  - $\forall k \in [N] \quad (\varphi_k)_{\boldsymbol{D}[d_1,\ldots,d_{n_i}][g_1,\ldots,g_N]} = g_k$

  C'est aussi équivalent à $g_i = (t_i)_{X_i,\boldsymbol{D}[g_1,\ldots,g_N]}$.
- La *sémantique* $sem_{\boldsymbol{D}}(\Sigma)$, notée $\Sigma_{\boldsymbol{D}}$, du système d'équations Σ dans une F-interprétation $\boldsymbol{D}$ est la *plus petite solution* $(\varphi_{1\boldsymbol{D}}, \ldots, \varphi_{N\boldsymbol{D}})$ *de* Σ *dans* $\boldsymbol{D}$.

Sémantique du plus petit point fixe.

- Pour tout $i \in [N]$, Δ_i est l'ensemble partiellement ordonné complet des fonctions continues de $D_i \to D_{\sigma(\varphi_i)}$, et $\widetilde{\Delta} = \Delta_1 \times \cdots \times \Delta_N$ l'ensemble partiellement ordonné complet produit.
- Soit $G : \widetilde{\Delta} \to \widetilde{\Delta}$ la fonction continue définie par $G(g_1, \ldots, g_N) = (\hat{g}_1, \ldots \hat{g}_N)$ telle que $\hat{g}_i(d_1, \ldots, d_{n_i}) = (t_i)_{\boldsymbol{D}[d_1,\ldots,d_{n_i}][g_1,\ldots,g_N]}$ pour tout $i \in [N]$ et tout $(d_1, \ldots, d_{n_i}) \in\ \in D_i$.

[8] C'est la dénomination employée par exemple par Berry et Lévy [BL79], Kott [Kot85], etc. En revanche, Courcelle [Cou90a] appelle *schéma de programme applicatif récursif* un couple $S = (\Sigma, t)$ où Σ est un système d'équations et $t \in M(F \cup \Phi, X)$. Le terme t joue le rôle d'une *procédure principale* de variables $Var(t)$, par opposition aux *fonctions auxiliaires* représentées par Σ. Il revient au même d'ajouter une équation $\varphi_0(x_1, \ldots, x_{n_0}) = t$ à Σ, où $\varphi_0 \notin \Phi$ est une nouvelle inconnue et $Var(t) = \{x_1, \ldots, x_{n_0}\}$.

Dans [Cou86, §7], Courcelle définit plus formellement les notions de *sous-systèmes* et d'*inconnues auxiliaires*. On peut ainsi représenter la *visibilité (scope) des procédures* d'un langage de programmation directement au niveau des schémas de programme. En pratique, cela revient à restreindre les fonctions dénotées par Σ à un sous-ensemble de $\{\varphi_{1\boldsymbol{D}}, \ldots, \varphi_{N\boldsymbol{D}}\}$. Nous ne faisons pas explicitement ce raffinement ici afin de simplifier la présentation. Cette notion sera néanmoins développée au chapitre 6 (cf. §5.5.1).

- Proposition : Un n-uplet $(g_1, \ldots, g_N)$ de $\widetilde{\Delta}$ est solution de Σ dans $\boldsymbol{D}$ ssi il est solution de $y = G(y)$ dans $\widetilde{\Delta}$.
- Proposition : $\Sigma_{\boldsymbol{D}} = \mathrm{lfp}(G) = G{\uparrow}\omega$ et $\Sigma_{\boldsymbol{D}} = (\Sigma_{\boldsymbol{M}_\Omega^\infty(F,X)})_{\boldsymbol{D}}$.

Sémantique opérationnelle.

- On note aussi Σ le système de réécriture $\{\langle\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) \to t_i\rangle \mid i \in [N]\}$.
- On note Ω le système de réécriture $\{\langle\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) \to \Omega_{s_i}\rangle \mid i \in [N]\}$.
- Soit $U(\boldsymbol{D})$ l'ensemble des paires (t, t') de $M_\Omega(F, X \cup D) \times M_\Omega(F, X \cup D)$ telles que :
  - $t = f(t_1, \ldots, t_n)$ avec $f \in F_n$,
  - pour tout $i \in [n]$, $t', t_i \in X \cup D$,
  - le terme t est X-linéaire (aucune variable de X n'apparaît plusieurs fois dans t),
  - $Var(t') \subset Var(t)$, c.-à-d. si $t' = x \in X$, alors x apparaît dans t.

  Soit $R(\boldsymbol{D}) \subset U(\boldsymbol{D})$ l'ensemble des règles de réécriture $\langle t \to t'\rangle$ de $U(\boldsymbol{D})$ *validées par* $\boldsymbol{D}$, c'est-à-dire telles que $t_{\boldsymbol{D}} = t'_{\boldsymbol{D}}$.
- Pour tout ensemble de régles de réécriture $R \subset M_\Omega(F, X \cup D) \times M_\Omega(F, X \cup D)$ et pour tout $t \in M_\Omega(F \cup \Phi, D)$, on note $Der_{\Sigma,R}(t) = \{d \in D \mid t \to^*_{\Sigma,\Omega,R} d\}$.
- Proposition : $Der_{\Sigma,R(\boldsymbol{D})}(t)$ est un ensemble dirigé pour tout $t \in M_\Omega(F \cup \Phi, D)$; il admet donc une borne supérieure $\sup(Der_{\Sigma,R(\boldsymbol{D})}(t))$, notée $t_{\Sigma,\boldsymbol{D}}$. On a aussi $Der_{\Sigma,R(\boldsymbol{D})}(t) = \{t'_{\boldsymbol{D}} \mid t' \in M_\Omega(F, D),\ t \to^*_{\Sigma,\Omega} t'\}$.
- Pour tout $t \in M(F \cup \Phi, X)_s$ et $X' = (x_1, \ldots, x_n) \in X^*$ tel que $Var(t) \subset X'$, on note $t_{X',\Sigma,\boldsymbol{D}}$ la fonction de $D_{\sigma(x_1)} \times \cdots \times D_{\sigma(x_n)} \to D_s$ définie par $t_{X',\Sigma,\boldsymbol{D}}(d_1, \ldots, d_n) = (t[x_1/d_1, \ldots, x_n/d_n])_{\Sigma,\boldsymbol{D}}$.
- Proposition : la plus petite solution de Σ dans $\boldsymbol{D}$ est $\Sigma_{\boldsymbol{D}} = (\varphi_i(x_1, \ldots, x_{n_i})_{X_i,\Sigma,\boldsymbol{D}})_{i \in [N]}$.
- Pour tout $t \in M(F \cup \Phi, X)$, on note $L(\Sigma, t) = \{t' \in M_\Omega(F, X) \mid t \to^*_{\Sigma,\Omega} t'\}$. L'ensemble $L(\Sigma, t)$ est dirigé dans l'algèbre ordonnée $\boldsymbol{M}_\Omega(F \cup X)$. On pose $T(\Sigma, t) = \sup(L(\Sigma, t)) = t_{\Sigma,\boldsymbol{M}_\Omega^\infty(F,X)}$ dans la F-algébre libre continue $M_\Omega^\infty(F, X)$. Proposition : $T(\Sigma, t)_{\boldsymbol{D}} = t_{\Sigma,\boldsymbol{D}}$.

Cette présentation de la sémantique opérationnelle est donnée par Courcelle dans [Cou90a]. On y trouve également une formulation plus simple dans le cas particulier d'une interprétation discrète.

Règles sémantiques.

Les règles de $R(\boldsymbol{D})$ ne suffisent cependant pas pour modéliser certains aspects de l'exécution (cf. §2.5.8, 2.5.9). Elles entraînent aussi parfois des choix moins naturels pour la syntaxe algébrique, version aplatie de la syntaxe abstraite, ou pour l'interprétation, qui comporte davantage de domaines fonctionnels.

Raoult et Vuillemin emploient des *règles de simplification* [RV80] plus générales que celles de $R(\boldsymbol{D})$, mais également insuffisantes car en fait notre objectif est différent ; nous ne cherchons pas à *exprimer* la sémantique des schémas ou à *implémenter* un ensemble de régles de réductions, mais à *modéliser* une implémentation déjà existante à l'aide de dérivations. C'est pourquoi nous étendons $R(\boldsymbol{D})$ et les règles de simplifications à $V(\boldsymbol{D})$, ensemble de toutes les équations sur $M_\Omega(F, X \cup D)$ qui sont *validées*[9] par l'interprétation $\boldsymbol{D}$.

[9] Raoult et Vuillemin notent [RV80] que leurs conditions explicites rejettent l'emploi de règles « parallèles » telles que $\langle \mathsf{if}(x, y, y) \to y\rangle$ ou $\langle \mathsf{sub}(x, x) \to 0\rangle$, qui compromettent la confluence des dérivations et qui n'admettent pas d'implémentation efficace [Vui74]. En fait, la condition de validité dans l'interprétation, dont ils font également usage, exclut naturellement et automatiquement ce type de règles : $\mathsf{if}(\Omega, y, y)_{\boldsymbol{D}} = \perp \neq y$ et $\mathsf{sub}(\Omega, \Omega)_D = \perp \neq 0$.

Définition 2.5. (Règles de simplification étendues)
Soit $\boldsymbol{D}$ une F-interprétation. On définit :

- $V(\boldsymbol{D})$ est l'ensemble des règles $\langle t \to t' \rangle$ de $M_\Omega(F, X \cup D) \times M_\Omega(F, X \cup D)$ tq $t_{\boldsymbol{D}} = t'_{\boldsymbol{D}}$.
- $V^\dagger(\boldsymbol{D})$ est l'ensemble des règles $\langle t \to t' \rangle$ de $M_\Omega(F, X \cup D) \times M_\Omega(F, X \cup D)$ tq $t_{\boldsymbol{D}} = t'_{\boldsymbol{D}}$ et $t \notin D$.

Notons que $R(\boldsymbol{D}) \subset V^\dagger(\boldsymbol{D}) \subset V(\boldsymbol{D})$.

Proposition 2.1. (Propriété des règles de simplification étendues)
Soient Σ un F-schéma, $\boldsymbol{D}$ une F-interprétation, et R un ensemble de règles tel que $R(\boldsymbol{D}) \subset R \subset V(\boldsymbol{D})$. Alors $\forall t \in M_\Omega(F \cup \Phi, D) \quad Der_{\Sigma,R}(t) = Der_{\Sigma,R(\boldsymbol{D})}(t)$.

Démonstration. L'inclusion $R(\boldsymbol{D}) \subset R$ fait que $Der_{\Sigma,R(\boldsymbol{D})}(t) \subset Der_{\Sigma,R}(t)$. La réciproque s'appuie sur les points suivants :

(1) On a $\to^*_{\Sigma,\Omega,R} = \to^*_{\Sigma,\Omega} \to^*_R$ grâce à la proposition 5.3 car Σ est linéaire à gauche (cf. §5.4), $R \subset V(\boldsymbol{D})$ est linéaire à droite donc $R^{-1} \subset V(\boldsymbol{D})^{-1}$ est linéaire à gauche, et $Crit(\Sigma \cup \Omega, V(\boldsymbol{D})^{-1}) = Crit(V(\boldsymbol{D})^{-1}, \Sigma \cup \Omega) = \varnothing$.

(2) Pour tout $t \in M_\Omega(F, D)$, si $t \to^*_R d \in D$, alors $t_{\boldsymbol{D}} = d$ par récurrence sur la longueur de la dérivation.

(3) Pour tout $t \in M_\Omega(F, D)$, si $t_{\boldsymbol{D}} = d \in D$ alors, par récurrence sur la structure de t, il existe une dérivation $t \to^*_{R(\boldsymbol{D})} d$.

Pour toute dérivation $t \to^*_{\Sigma,\Omega,R} d \in D$, il existe $t' \in M_\Omega(F, D)$ tel que $t \to^*_{\Sigma,\Omega} t' \to^*_R d$ par (1) et $t'_{\boldsymbol{D}} = d$ par (2). Donc il existe une dérivation $t' \to_{R(\boldsymbol{D})} d$ par (3), et $t \to^*_{\Sigma,\Omega} t' \to^*_{R(\boldsymbol{D})} d$, c'est-à-dire $t \in Der_{\Sigma,R(\boldsymbol{D})}(t)$. $\square$

Parmi les définitions et propriétés ci-dessus, celles qui découlent de $R(\boldsymbol{D})$ s'appliquent également à $R \subset V(\boldsymbol{D})$ car seule intervient la borne supérieure de $Der_{\Sigma,R}(t) = Der_{\Sigma,R(\boldsymbol{D})}(t)$. Cette proposition nous permet donc d'utiliser toutes les règles de $V(\boldsymbol{D})$, plus générales. Réciproquement, ce qui concerne $V(\boldsymbol{D})$ dans la suite s'applique en particulier à $R(\boldsymbol{D})$ et aux règles de simplification.

2.5.3 Spécification.

La *sémantique* $sem_{\boldsymbol{D}}$ (cf. §2.5.2, solution d'un schéma) associe une fonction à chaque inconnue d'un schéma de programme. Suivant qu'un programme $p \in \mathcal{P}$ définit une fonction *unique* ou une *bibliothèque de fonctions*, un nom d'inconnue $\varphi \in \boldsymbol{\Phi}$ est incorporé soit aux schémas $\Sigma \in Sch(F)$, soit aux données $d \in \boldsymbol{D}$ (cf. §2.1.1).

La spécification d'un langage de programmation $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ en termes de schémas récursifs suit le modèle de la compilation (cf. §2.1.3). Il faut donner :

- une syntaxe algébrique à l'aide d'une signature F,
- une F-interprétation $\boldsymbol{D}$,
- une traduction $(T_\mathcal{P}, T_\mathcal{D})$ de $\mathcal{P} \times \mathcal{D}$ dans $Sch(F) \times \boldsymbol{\Phi} \times \boldsymbol{D}$,
- une traduction $T_\mathcal{R}$ de $\boldsymbol{D}$ dans $\mathcal{R}$,
- éventuellement, une stratégie de réduction ϱ.

Le dernier point est lié à l'aspect opérationnel : la sémantique des schémas de programme donnée à la section précédente correspond par exemple au principe de l'*appel par nom* des fonctions $\varphi(exp_1, \ldots, exp_n)$. Si la sémantique du langage emploie un autre mode d'évaluation telle que l'*appel par valeur*, il faut également le préciser. C'est l'objet d'une *stratégie de réduction* (cf. §2.5.4).

Sémantiques et traductions sont reliées ainsi :

$$\begin{tikzcd}
\mathcal{P} \arrow[d, "T_{\mathcal{P}}"'] & \times & \mathcal{D} \arrow[r, "L"] \arrow[d, "T_{\mathcal{D}}"'] & \mathcal{R} \\
Sch(F) & \times \, \boldsymbol{\Phi} \, \times & \boldsymbol{D} \arrow[r, "sem_{\boldsymbol{D}}"] & \boldsymbol{D} \arrow[u, "T_{\mathcal{R}}"']
\end{tikzcd}$$

Lorsque $\boldsymbol{D}$ est une interprétation discrète, il est fréquent que $\boldsymbol{D}$ et $\mathcal{R}$ coïncident à l'élément $\perp$ près, qui modélise la non-terminaison : les $(p, d) \in \mathcal{P} \times \mathcal{D}$ tels que $T_{\mathcal{R}}(sem_{\boldsymbol{D}}\ T_{\mathcal{P}}(p)\ T_{\mathcal{D}}(d)) = \perp$ n'appartiennent pas à $Dom(L)$. Il n'est alors pas nécessaire de donner la traduction $T_{\mathcal{R}}$. C'est le cas dans l'exemple suivant pour BOOL.

Syntaxe algébrique. Une *syntaxe algébrique* de BOOL est définie par :

- $\mathcal{S} = \{\text{bool}\}$
- $X = \{x_{var} : \text{bool} \mid var \in \text{Var}\}$
- $F = \{\mathsf{true} : \text{bool}\ ,\ \mathsf{false} : \text{bool}\ ,\ \mathsf{if} : \text{bool} \times \text{bool} \times \text{bool} \to \text{bool}\}$
- $\Phi \subset \boldsymbol{\Phi} = \{\varphi_{fun} : \text{bool}^* \to \text{bool} \mid fun \in \text{Fun}\}$

Par ailleurs, nous notons par exemple **nand** une inconnue φ_{nand}, et en minuscules italiques $x, y \ldots$ des variables comme $x_{\mathsf{X}}, x_{\mathsf{Y}} \ldots$

La différence majeure entre la *syntaxe abstraite* qui était sous-entendue dans la description syntaxique de BOOL (cf. §2.2.1) et la *syntaxe algébrique* donnée ci-dessus se situe au niveau des appels fonctionnels. Pour représenter « $fun(exp_1, \ldots, exp_n)$ », on a dans le premier cas un opérateur explicite d'appel « $\mathsf{call}(fun, exp_1, \ldots, exp_n)$ ». Dans le second, on emploie une inconnue « $\varphi_{fun}(t_1, \ldots, t_n)$ ». Cette différence sera étudiée plus en détail à la section §2.5.7.

Interprétation. On définit une F-interprétation $\boldsymbol{D} = \langle D, \leqslant, \perp, (f_{\boldsymbol{D}})_{f \in F}\rangle$ par :

- $D = \text{BoolVal}_\perp = \text{BoolVal} \cup \{\perp\} = \{true, false, \perp\}$
- Pour tous $v, v' \in \text{BoolVal}_\perp,\ v \leqslant v'$ ssi $(v = v'$ ou $v = \perp)$
- $\mathsf{true}_{\boldsymbol{D}} = true$
- $\mathsf{false}_{\boldsymbol{D}} = false$
- $\mathsf{if}_{\boldsymbol{D}}(v_1, v_2, v_3) = \begin{cases} v_2 & \text{si } v_1 = true \\ v_3 & \text{si } v_1 = false \\ \perp & \text{si } v_1 = \perp \end{cases}$

C'est une interprétation discrète.

Traduction dans les schémas. La figure 2.8 donne une traduction des programmes $T : \text{Dec} \to Sch(F)$. La condition syntaxique sur les fonctions définies par « $fun(var_1, \ldots, var_n)\ =\ exp$ » (cf. §2.2.1) garantit que la condition $Var(T(exp)) \subset \{x_{var_1}, \ldots, x_{var_n}\}$ sur les schémas (cf. §2.5.2) est bien vérifiée. La traduction des données est définie par la fonction :

- T' : Fun $\times$ BoolVal* $\rightarrow \mathbf{\Phi} \times$ BoolVal$^*_\perp$ telle que $T'(fun, (v_1, \ldots, v_n)) = (\varphi_{fun}, (v_1, \ldots, v_n))$

La traduction T'' : BoolVal$_\perp \rightarrow$ BoolVal vers les résultats est l'identité restreinte à BoolVal ; l'élément $\perp$ dénote une exécution qui ne se termine pas.

$$
\begin{array}{lcl}
T(\texttt{true}) & = & \mathsf{true} \\
T(\texttt{false}) & = & \mathsf{false} \\
T(var) & = & x_{var} \\
T(\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3) & = & \mathsf{if}(T(exp_1), T(exp_2), T(exp_3)) \\
T(fun\texttt{(}exp_1\texttt{,} \ldots \texttt{,} exp_n\texttt{)}) & = & \varphi_{fun}(T(exp_1), \ldots, T(exp_n)) \\
 & & \\
T(fun\texttt{(}var_1\texttt{,} \ldots \texttt{,} var_n\texttt{) = } exp) & = & \varphi_{fun}(x_{var_1}, \ldots, x_{var_n}) = T(exp) \\
T(funbind\ \langle\texttt{and } funbinds\rangle) & = & \{T(funbind)\}\ \langle \cup\, T(funbinds)\rangle \\
T(\texttt{fun } funbinds) & = & T(funbinds)
\end{array}
$$

Figure 2.8 : Traduction des programmes BOOL dans les schémas récursifs

Sémantique. Par exemple, la fonction `nand` définie par « `fun nand(X,Y) = if X then (if Y then false else true) else true` » (cf. §2.2.2) est traduite en :

$$\mathbf{nand}(x, y) = \mathsf{if}(x, \mathsf{if}(y, \mathsf{false}, \mathsf{true}), \mathsf{true})$$

Sa sémantique du plus petit point fixe, simple car le corps de **nand** ne comporte pas d'appels fonctionnels, est :

- Pour tous $x, y \in$ BoolVal$_\perp$ $\mathbf{nand}_{\boldsymbol{D}}(x, y) = \begin{cases} \mathit{true} & \text{si } x = \mathit{false} \text{ ou } (x = \mathit{true} \text{ et } y = \mathit{false}) \\ \mathit{false} & \text{si } x = \mathit{true} \text{ et } y = \mathit{true} \\ \perp & \text{sinon} \end{cases}$

Elle vérifie bien $\mathbf{nand}_{\boldsymbol{D}}(\mathit{true}, \mathit{false}) = \mathit{true}$, résultat que l'on peut aussi obtenir par la dérivation suivante :

$$\underline{\mathbf{nand}}(\mathit{true}, \mathit{false}) \rightarrow \underline{\mathsf{if}(\mathit{true}}, \mathsf{if}(\mathit{false}, \mathsf{false}, \mathsf{true}), \mathsf{true}) \rightarrow \underline{\mathsf{if}(\mathit{false}}, \mathsf{false}, \mathsf{true}) \rightarrow \underline{\mathsf{true}} \rightarrow \mathit{true}.$$

2.5.4 Stratégies et réductions.

La sémantique opérationnelle définie ci-dessus est un processus de calcul à la fois *non-déterministe* et *limite* : on détermine la valeur d'un terme t comme la borne supérieure des dérivations $t \rightarrow^*_{\Sigma,\Omega,R} t'$. Ce n'est pas à proprement parler une *exécution*.

Stratégie de réduction.

Le comportement opératoire d'un schéma de programme récursif est donné par une *stratégie de réduction* (computation rule) qui indique comment (dans quel ordre et à quelles occurrences des termes) effectuer en pratique ces dérivations. C'est un *algorithme* qui, étant donné un terme $t \in M(F \cup \Phi, D)$, produit un $t' \in M(F \cup \Phi, D)$ tel que $t \rightarrow_{\Sigma,\Omega,R} t'$, ou bien signale qu'un tel t' n'existe pas, lorsque t ne comporte pas d'*expression réductible* (redex).

Définition 2.6. (Stratégie de réduction)
Reprenons les notations la section §2.5.2 (sémantique opérationnelle). Soit R un ensemble de règles telles que $R(\boldsymbol{D}) \subset R \subset V^{\dagger}(\boldsymbol{D})$. Une *stratégie de réduction (en un pas)* est une application calculable $\varrho : M(F \cup \Phi, D) \to M(F \cup \Phi, D) \cup \{\dagger\}$ qui vérifie :

- $t \to_{\Sigma,\Omega,R} \varrho(t)$ ou $(t \not\to_{\Sigma,\Omega,R}$ et $\varrho(t) = \dagger)$ pour tout $t \in M(F \cup \Phi, D)$

L'absence d'*expression réductible* est symbolisée par le symbole « † ». Une stratégie de réduction *en plusieurs pas* vérifie : pour tout $t \in M(F \cup \Phi, D)$ $t \to^{+}_{\Sigma,\Omega,R} \varrho(t)$ ou $(t \not\to_{\Sigma,\Omega,R}$ et $\varrho(t) = \dagger)$. On définit ensuite :

- $\tilde{\varrho} : M(F \cup \Phi, D) \to M(F \cup \Phi, D)$ tq $\tilde{\varrho}(t) = \begin{cases} t & \text{si } \varrho(t) = \dagger \\ \varrho(t) & \text{sinon} \end{cases}$ pour tout $t \in M(F \cup \Phi, D)$
- $\omega : M(F \cup \Phi, X \cup D) \to M_{\Omega}(F, X \cup D)$ est la fonction qui substitue dans un terme t toutes les occurrences d'inconnues $\varphi \in \Phi$ par Ω : $\forall d \in D \quad \forall \varphi \in \Phi \quad \forall f \in F_n \quad \forall x \in X \quad \forall t_1, \ldots, t_n \in M(F \cup \Phi, X \cup D)$,
  - $\omega(x) = x$
  - $\omega(d) = d$
  - $\omega(\varphi(t_1, \ldots, t_n)) = \Omega$
  - $\omega(f(t_1, \ldots, t_n)) = f(\omega(t_1), \ldots, \omega(t_n))$

  Notons que $t \to^{*}_{\Omega} \omega(t)$.
- $\mathrm{h}_{\boldsymbol{D}} : M_{\Omega}(F, D) \to D$ est l'interprétation dans $\boldsymbol{D}$ des termes formés sur les signatures F et D : $\forall d \in D$ $\forall f \in F_n \quad \forall t_1, \ldots, t_n \in M_{\Omega}(F, D)$,
  - $\mathrm{h}_{\boldsymbol{D}}(d) = d$
  - $\mathrm{h}_{\boldsymbol{D}}(\Omega) = \perp_{\boldsymbol{D}}$
  - $\mathrm{h}_{\boldsymbol{D}}(f(t_1, \ldots, t_n)) = f_{\boldsymbol{D}}(\mathrm{h}_{\boldsymbol{D}}(t_1), \ldots, \mathrm{h}_{\boldsymbol{D}}(t_n))$

  C'est le prolongement à $M_{\Omega}(F, D)$ de l'application de $M_{\Omega}(F) \to D$ qui à t associe $t_{\boldsymbol{D}}$. Notons également que $t \to^{*}_{R} \mathrm{h}_{\boldsymbol{D}}(t)$ car $R(\boldsymbol{D}) \subset R$.
- $t_{\Sigma,\boldsymbol{D},\varrho} = \sup_{n \in \mathrm{IN}}(\mathrm{h}_{\boldsymbol{D}}(\omega(\tilde{\varrho}^{n}(t))))$.

Le terme $t_{\Sigma,\boldsymbol{D},\varrho}$ est la valeur de t *calculée par* ϱ dans l'interprétation $\boldsymbol{D}$. Comme pour la sémantique opérationnelle, si $t \in M(F \cup \Phi, X \cup D)$ avec $Var(t) \subset \{x_1, \ldots, x_n\} = X_n$, alors $t_{X_n,\Sigma,\boldsymbol{D},\varrho}$ désigne la fonction qui à $(d_1, \ldots, d_n) \in D^n$ associe $(t[x_1/d_1, \ldots, x_n/d_n])_{\Sigma,\boldsymbol{D},\varrho}$.

Il faut s'assurer que le terme $t_{\Sigma,\boldsymbol{D},\varrho}$ défini ci-dessus a bien un sens :

Démonstration. On démontre par récurrence sur la longueur de la dérivation $t \to^{*}_{\Sigma,R} \tilde{\varrho}(t)$ que $\mathrm{h}_{\boldsymbol{D}}(\omega(t)) \leqslant \mathrm{h}_{\boldsymbol{D}}(\omega(\tilde{\varrho}(t)))$. Il y a trois cas, suivant que l'on fait un pas de dérivation par Σ, par Ω ou par R. Pour tout $t, t' \in M(F \cup \Phi, D)$,

- Si $t \to_{\Sigma} t'$ alors $\omega(t) \leqslant \omega(t')$ dans $\boldsymbol{M}_{\Omega}(F, X \cup D)$, donc $\mathrm{h}_{\boldsymbol{D}}(\omega(t)) \leqslant \mathrm{h}_{\boldsymbol{D}}(\omega(t'))$ par monotonie.
- Si $t \to_{\Omega} t'$ alors $\omega(t) = \omega(t')$, donc $\mathrm{h}_{\boldsymbol{D}}(\omega(t)) = \mathrm{h}_{\boldsymbol{D}}(\omega(t'))$.
- Si $t \to_{u,r} t'$ avec $r \in R$, alors $t/u \to_{\epsilon,r} t'/u$ donc $\omega(t/u) \to_{\epsilon,r} \omega(t'/u)$ car $Crit(\Omega, R) = \varnothing$, donc $\omega(t/u)_{\boldsymbol{D}} = \omega(t'/u)_{\boldsymbol{D}}$ car $r \in R$, donc $\omega(t)_{\boldsymbol{D}} = \omega(t')_{\boldsymbol{D}}$ car $t[u \leftarrow \square] = t'[u \leftarrow \square]$, c.-à-d. $\mathrm{h}_{\boldsymbol{D}}(\omega(t)) = \mathrm{h}_{\boldsymbol{D}}(\omega(t'))$.

Par conséquent, $(\mathrm{h}_{\boldsymbol{D}}(\omega(\tilde{\varrho}^{n}(t))))_{n \in \mathrm{IN}}$ est une chaîne croissante dans l'algèbre continue $\boldsymbol{D}$; elle admet donc une borne supérieure $t_{\Sigma,\boldsymbol{D},\varrho}$. □

Cette définition étend celle que donne Courcelle [Cou90a, §3.5] pour les interprétations discrètes. Elle diffère de celles de Berry et Lévy [BL79], ou celle de Kott [Kot80], qui se focalisent sur la réécriture aux occurrences de fonctions récursives, c'est-à-dire sur la relation $\rightarrow_\Sigma$.

L'extension aux interprétations continues est nécessaire en pratique car les interprétations discrètes ne permettent pas de représenter toutes les constructions d'un langage de programmation. Par exemple, elle n'autorisent pas la manipulation en tant que valeurs, de produits cartésiens, de fonctions, et de suites infines. En particulier, elles ne peuvent se plier aux constructions des sections §2.5.8 et §2.5.9, qui nécessitent l'emploi d'objets fonctionnels.

Les stratégies de réduction, telles qu'elles sont formalisées à la définition 2.6, vérifient la proposition suivante.

Proposition 2.2. (Propriété des stratégies de réduction)
Soient Σ un F-schéma d'inconnues Φ, $\boldsymbol{D}$ une F-interprétation et ϱ une stratégie de réduction. Pour tout $t \in M(F \cup \Phi, D)$ on a :

- $\varrho(t) = \dagger$ ssi $t \in D$ ssi $t_{\Sigma,\boldsymbol{D},\varrho} = t$
- $t_{\Sigma,\boldsymbol{D},\varrho} \leqslant t_{\Sigma,\boldsymbol{D}}$

Démonstration.

- Si $\varrho(t) = \dagger$ alors $t \not\rightarrow_{\Sigma,R}$ donc $Occ_F(t) = \varnothing$ car $R(\boldsymbol{D}) \subset R$ et $Occ_\Phi(t) = \varnothing$. Par conséquent $t \in D$. Réciproquement si $t \in D$ alors $t \not\rightarrow_R$ car $R \subset V^\dagger(\boldsymbol{D})$ et $t \not\rightarrow_\Sigma$ donc $\varrho(t) = \dagger$.
  Si $t \in D$ alors pour tout $n \in \mathbb{N}$ $\mathrm{h}_{\boldsymbol{D}}(\omega(\tilde{\varrho}^n(t))) = \mathrm{h}_{\boldsymbol{D}}(\omega(t)) = \mathrm{h}_{\boldsymbol{D}}(t) = t$ donc $t_{\Sigma,\boldsymbol{D},\varrho} = t$. Réciproquement si $t_{\Sigma,\boldsymbol{D},\varrho} = t$ alors $t \in D$ par définition.
- Pour tout $t \in M(F \cup \Phi, D)$ pour tout $n \in \mathbb{N}$ $t \rightarrow^*_{\Sigma,R} \tilde{\varrho}^n(t) \rightarrow^*_\Omega \omega(\tilde{\varrho}^n(t)) \rightarrow^*_R \mathrm{h}_{\boldsymbol{D}}(\omega(\tilde{\varrho}^n(t))) \in D$ donc $t_{\Sigma,R,\varrho} \in Der_{\Sigma,R}(t)$. Or $t_{\Sigma,R} = \sup(Der_{\Sigma,R}(t))$ donc $t_{\Sigma,\boldsymbol{D},\varrho} \leqslant t_{\Sigma,\boldsymbol{D}}$. $\square$

Dérivations finies.

Dans le cas d'une interprétation discrète, la suite croissante $(\mathrm{h}_{\boldsymbol{D}}(\omega(\tilde{\varrho}^n(t))))_{n\in\mathbb{N}}$ ne peut prendre au plus que deux valeurs distinctes, $\perp$ ou $d \in D \setminus \{\perp\}$. Par conséquent,

- soit elle atteint un $d \in D \setminus \{\perp\}$ en un nombre fini d'étapes : c'est nécessairement le résultat final,
- soit elle reste stationnaire à $\perp$:
  - ou bien les inconnues φ ne disparaissent jamais de $(\tilde{\varrho}^n(t))_{n\in\mathbb{N}}$: l'exécution ne se termine pas,
  - ou bien elles disparaissent de $(\tilde{\varrho}^n(t))_{n\in\mathbb{N}}$ à partir d'un certain rang : le résultat final est $\perp$.

  Ce dernier cas ne se produit pas si $\perp$ n'est employé dans les fonctions de $\boldsymbol{D}$ que pour dénoter une valeur indéfinie et non les cas d'erreur (division par zéro, etc.).

La situation est très différente dans une interprétation continue car il peut être nécessaire de « passer à la limite » pour obtenir la valeur $t_{\Sigma,\boldsymbol{D},\varrho}$. Néanmoins, les éléments *finis maximaux* de $\boldsymbol{D}$ sont calculés par une dérivation finie [Cou90a, §4.2].

Une stratégie de réduction ne définit donc pas nécessairement un procédé de calcul *effectif* dans le cas d'une algèbre continue : la *valeur effectivement calculée* par une stratégie de réduction est une valeur calculée par dérivation finie.

Définition 2.7. (Valeur effectivement calculée par une stratégie de réduction)
Reprenons les notations de la définition 2.6. Pour tout $t \in M(F \cup \Phi, D)$,

- l'exécution de t par la stratégie ϱ *se termine* ssi il existe $n \in \mathbb{N}$ $\varrho(\tilde{\varrho}^n(t)) = \dagger$.

La valeur de t *effectivement* calculée par la stratégie ϱ est alors $t_\varrho = \tilde{\varrho}^n(t) = t_{\Sigma,\boldsymbol{D},\varrho} \in D$ (prop. 2.2). Dans le cas contraire, on dit que l'*exécution ne se termine pas* et l'on pose $t_\varrho = \perp$.

Notons que pour tout $t \in M(F \cup \Phi, D)$, on a $t_\varrho \leqslant t_{\Sigma,\boldsymbol{D},\varrho} \leqslant t_{\Sigma,\boldsymbol{D}}$.

Exemples.

Voici deux exemples de stratégies de réduction sur BOOL. La stratégie ϱ_1, donnée figure 2.9, est basée sur l'*appel par nom*. La stratégie de réduction ϱ_2, donnée figure 2.10, emploie l'*appel par valeur*. Elle ne diffère de ϱ_1 que sur l'appel fonctionnel $\varphi(exp_1, \ldots, exp_n)$.

$$
\begin{array}{lcl}
\varrho_1(true) = \varrho_1(false) & = & \dagger \\
\varrho_1(\mathsf{true}) & = & true \\
\varrho_1(\mathsf{false}) & = & false \\
\varrho_1(\mathsf{if}(t_1, t_2, t_3)) & = & \begin{cases} t_2 & \text{si } t_1 = true \\ t_3 & \text{si } t_1 = false \\ \mathsf{if}(\varrho_1(t_1), t_2, t_3) & \text{sinon} \end{cases} \\
\varrho_1(\varphi(t_1, \ldots, t_n)) & = & t[x_1/t_1, \ldots, x_n/t_n] \quad \text{pour} \quad \langle \varphi(x_1, \ldots, x_n) = t \rangle \in \Sigma
\end{array}
$$

Figure 2.9 : Appel par nom dans BOOL

$$
\begin{array}{lcl}
\varrho_2(true) = \varrho_2(false) & = & \dagger \\
\varrho_2(\mathsf{true}) & = & true \\
\varrho_2(\mathsf{false}) & = & false \\
\varrho_2(\mathsf{if}(t_1, t_2, t_3)) & = & \begin{cases} t_2 & \text{si } t_1 = true \\ t_3 & \text{si } t_1 = false \\ \mathsf{if}(\varrho_2(t_1), t_2, t_3) & \text{sinon} \end{cases} \\
\varrho_2(\varphi(t_1, \ldots, t_n)) & = & \begin{cases} \varphi(\varrho_2(t_1), \ldots, t_n) & \text{si } \varrho_2(t_1) \neq \dagger, \text{ sinon} \\ \ldots & \\ \varphi(t_1, \ldots, \varrho_2(t_n)) & \text{si } \varrho_2(t_n) \neq \dagger, \text{ sinon} \\ t[x_1/t_1, \ldots, x_n/t_n] & \text{pour } \langle \varphi(x_1, \ldots, x_n) = t \rangle \in \Sigma \end{cases}
\end{array}
$$

Figure 2.10 : Appel par valeur dans BOOL

Ces deux stratégies coïncident sur la dérivation de $\mathbf{nand}(true, false)$, qui est celle que nous avons déjà donné ci-dessus (cf. §2.5.3, sémantique). Pour voir leur différence, considérons le programme BOOL suivant :

```
fun phi(X,Y) = if X then true else phi(true,phi(false,Y))
```

Il est traduit par T en le schéma de programme Σ suivant :

$$\varphi(x,y) = \mathsf{if}(x, \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, y)))$$

L'expression `phi` $(false, true)$ est dérivée ainsi par ϱ_1 (les expressions réduites par ϱ_1 sont soulignées) :

$$\begin{array}{rcl} \underline{\varphi}(false, true) & \to & \underline{\mathsf{if}}(false, \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, true))) \\ & \to & \underline{\varphi}(\mathsf{true}, \varphi(\mathsf{false}, true)) \\ & \to & \mathsf{if}(\underline{\mathsf{true}}, \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, \varphi(\mathsf{false}, true)))) \\ & \to & \underline{\mathsf{if}}(true, \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, \varphi(\mathsf{false}, true)))) \\ & \to & \underline{\mathsf{true}} \\ & \to & true. \end{array}$$

Plus généralement, $\varphi_{\Sigma,\boldsymbol{D}} = \varphi_{\Sigma,\boldsymbol{D},\varrho_1}$ est la fonction constante de valeur $true$. En revanche, la dérivation selon la stratégie de réduction ϱ_2 ne se termine pas (et donc $\varphi(false, true)_{\varrho_2} = \bot$) :

$$\begin{array}{rcl} \underline{\varphi}(false, true) & \to & \underline{\mathsf{if}}(false, \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, true)) \\ & \to & \varphi(\mathsf{true}, \varphi(\underline{\mathsf{false}}, true)) \\ & \to & \varphi(\mathsf{true}, \underline{\varphi}(false, true)) \\ & \stackrel{*}{\to} & \varphi(\mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{true}, \ldots \underline{\varphi}(false, true) \ldots))) \end{array}$$

On a même en fait $\varphi(false, true)_{\Sigma,\boldsymbol{D},\varrho_2} = \bot$. Cette différence est liée à la notion de *correction* d'une stratégie de réduction.

Correction des stratégies de réduction.

Nous avons vu que pour tout $t \in M(F \cup \Phi)$, la relation $t_{\Sigma,\boldsymbol{D},\varrho} \leqslant t_{\Sigma,\boldsymbol{D}}$ est toujours vérifiée (prop. 2.2). En revanche, la relation réciproque ne l'est pas nécessairement.

Définition 2.8. (Correction des stratégies de réduction)
Une stratégie de réduction ϱ est *correcte* ssi $t_{\Sigma,\boldsymbol{D},\varrho} = t_{\Sigma,\boldsymbol{D}}$ pour tout $t \in M(F \cup \Phi)$.

Par exemple, pour tout $\Sigma \in Sch(F)$ et $t \in M(F \cup \Phi)$, les deux stratégies de réduction ϱ_1 et ϱ_2 données précédemment vérifient $t_{\Sigma,\boldsymbol{D},\varrho_2} \leqslant t_{\Sigma,\boldsymbol{D},\varrho_1} = t_{\Sigma,\boldsymbol{D}}$. La stratégie ϱ_1 est correcte, ϱ_2 ne l'est pas.

Il existe diverses stratégies correctes dans l'interprétation libre [Vui74, DS76, BL79, HL91], et notamment :

- l'*appel par nom* (call by name) qui réduit l'expression réductible la plus à gauche et la plus à l'extérieure,
- l'*appel par nom parallèle* qui réduit en parallèle les expressions réductibles les plus à l'extérieur,
- la *substitution totale* (full substitution) qui récrit simultanément toutes les occurrences des inconnues.

En revanche, l'*appel par valeur* (call by value) n'est pas toujours correct.

Les exemples du *ou parallèle* [HL91] et de la *multiplication parallèle* [BL79] montrent que la situation est plus complexe si l'on veut construire de manière systématique des stratégies séquentielles (qui en particulier ne nécessitent pas l'évaluation de termes en parallèle) effectives correctes dans une interprétation $\boldsymbol{D}$ quelconque. Notre formulation ne prétend pas apporter une solution à ce problème.

De manière générale, les stratégies de réduction correctes vont de l'extérieur des termes vers l'intérieur. Ce style de réduction a le défaut de dupliquer des termes et d'engendrer ainsi des calculs redondants. Berry et Lévy [BL79] étudient des notions de réductions optimales dans l'interprétation libre en utilisant une forme de partage. Elles s'étendent à diverses classes d'interprétations moyennant des conditions de séquentialité, de stabilité, ou de projection [BL79]. Un autre type d'interprétation permet l'implémentation effective de la stratégie de l'appel par nécessité [HL91].

Notre but n'est toutefois pas d'*implémenter* effectivement ou efficacement les schémas de programme récursifs, mais de les utiliser pour *modéliser* une implémentation existante : c'est le langage qui fixe la *sémantique* (appel par nom, par valeur. . .), et c'est l'implémentation *visée* qui fixe le *comportement dynamique* à l'exécution (avec partage ou non. . .).

2.5.5 Machine d'exécution.

Comme nous l'avons fait pour la sémantique naturelle (cf. §2.4.3), une spécification en termes de schéma de programme munie d'une stratégie de réduction peut être considérée comme une machine d'exécution abstraite. Chaque règle de réécriture de $\Sigma \cup R(\boldsymbol{D}) \cup \Omega$ correspond à une opération élémentaire de cette machine, et exécuter un programme (l'arbre de syntaxe algébrique) consiste à le dériver selon la stratégie de réduction donnée, jusqu'à obtenir un terme de l'interprétation $\boldsymbol{D}$ (déf. 2.7) ; une exécution se termine ssi la dérivation associée est finie.

Si la spécification au moyen des schémas est le seul support de la sémantique et de l'exécution (cf. §2.3.3), sa pertinence suppose que chacune des règles de réécriture « reflète » un comportement dynamique dans l'implémentation visée. Cela repose sur les *choix* de traduction $(T_{\mathcal{P}}, T_{\mathcal{D}}, T_{\mathcal{R}})$ et d'interprétation $\boldsymbol{D}$, et sur la notion de *fonction séquentielle* [Vui74] pour la construction d'une stratégie de réduction.

Parce que cette hypothèse est à la fois très spécifique au langage traité, et relativement subjective lorsque l'implémentation n'est pas spécifiée, nous nous contentons de l'illustrer sur BOOL. Parmi les règles de $R(\boldsymbol{D})$, nous considérons en particulier les celles de la figure 2.11. On note r_φ une règle $\langle \varphi(x_1, \ldots, x_n) \rightarrow t \rangle$ d'un schéma Σ.

$$
\begin{array}{rcll}
\mathsf{true} & \rightarrow & \mathit{true} & (r_{\mathsf{true}}) \\
\mathsf{false} & \rightarrow & \mathit{false} & (r_{\mathsf{false}}) \\
\mathsf{if}(\mathit{true}, x, y) & \rightarrow & x & (r_{\mathsf{if}\,\mathit{true}}) \\
\mathsf{if}(\mathit{false}, x, y) & \rightarrow & y & (r_{\mathsf{if}\,\mathit{false}})
\end{array}
$$

Figure 2.11 : Règles de réécriture pour la sémantique de BOOL

Une règle comme $\langle \mathsf{if}(\mathit{true}, x, y) \rightarrow x \rangle$ suppose que la condition du `if` a été évaluée en premier (ici à true) et que l'on va ensuite évaluer la partie `then` (le terme qui instancie x). Cependant $R(\boldsymbol{D})$ contient aussi des règles comme $\langle \mathsf{if}(true, x, \mathit{false}) \to x \rangle$, qui n'ont aucun sens pour BOOL. Cette règle signifie opérationnellement que la partie `else` du `if` (le troisième argument) doit systématiquement être évaluée, alors même que la condition est vraie. De même, une règle comme $\langle \mathsf{if}(true, \mathit{false}, y) \to \mathit{false} \rangle$ repousse conceptuellement le test effectif de la condition après l'évaluation de la partie `then`. Enfin, il importe peu que $R(\boldsymbol{D})$ soit un ensemble infini car nous ne nous préoccupons pas d'implementer explicitement la sémantique à l'aide de systèmes de réécriture mais de modéliser une exécution réelle.

2.5.6 Modèle d'exécution.

Comme la trace en sémantique naturelle est une vue de l'arbre de preuve (cf. §2.4.4), la trace d'exécution dans les schémas de programme est une *vue* de la *dérivation* dans une sémantique opérationnelle avec une stratégie de réduction. Nous donnons aussi un autre type de trace au chapitre suivant (cf. §3.2.6, interprétation).

Trace d'exécution.

Dans le cas général, la *trace d'exécution* est la *dérivation* elle-même, c'est-à-dire la suite des règles, occurrences et valeurs des variables (les substitutions) qui caractérisent chacune des réductions de la dérivation (cf. §N.17).

Pour une trace plus abstraite (cf. §2.3.2), on omet l'occurrence et la valeur des variables lorsqu'elles n'influencent pas le comportement dynamique modélisé par la règle. Le nom même des fonctions φ appelées n'a pas d'importance non plus. Si la nature des paramètres et leur nombre d'occurrences dans le corps de la fonction n'influent pas sur le comportement dynamique du passage d'arguments, la donnée de l'arité suffit à caractériser l'appel fonctionnel.

Les modèles $M_{\text{Schéma}}(\varrho_1)$ et $M_{\text{Schéma}}(\varrho_2)$ qui correspondent aux stratégies ϱ_1 et ϱ_2 précédentes ne modélisent pas convenablement l'implémentation de BOOL donnée à la section §2.2. Non seulement la stratégie ϱ_1 duplique des calculs, mais elle définit aussi une sémantique différente ; ϱ_1 est correcte à la différence de l'implémentation (et de la spécification en sémantique naturelle, §2.4.2) qui emploie l'appel par valeur : $(\mathit{false}, true)$ n'appartient pas au domaine de définition de $Bool(\texttt{phi})$ (cf. §2.5.4, exemples), alors que $\varphi(\mathit{false}, true)_{\varrho_1} = true$. D'autre part, la substitution $\varrho(\varphi(t_1, \ldots, t_n)) = t[x_1/t_1, \ldots, x_n/t_n]$, commune à ϱ_1 et ϱ_2, fait disparaître toute information sur les variables. Par exemple, la dérivation suivante

$$\underline{\varphi}(true, true) \to \underline{\mathsf{if}}(true, \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, true))) \to \underline{\mathsf{true}} \to true.$$

a pour trace $\langle\!\langle r_\varphi, r_{\mathsf{if}\,true}, r_{\mathsf{true}} \rangle\!\rangle$; elle ne modélise pas l'accès dynamique à un environnement pour connaître la valeur de x, image de la variable `X` dans les schémas de programme, au moment où la condition du `if` est évaluée.

Trace d'exécution des variables.

Pour ne pas perdre cette information, une solution est d'introduire dans la signature F un nouvel opérateur « $\mathsf{var} : \mathrm{bool} \to \mathrm{bool}$ ». La traduction d'une variable du programme en sa syntaxe algébrique (fig. 2.8) est modifiée de la manière suivante.

$$T(var) \quad = \quad \mathsf{var}(var)$$

Autrement dit, on remplace $\langle\varphi(x_1,\ldots,x_n) = t\rangle$ par $\langle\varphi(x_1,\ldots,x_n) = t[x_1/\mathsf{var}(x_1),\ldots,x_n/\mathsf{var}(x_n)]\rangle$. La sémantique de l'opérateur var dans $\boldsymbol{D}$ est la fonction identité ; son seul rôle est de marquer explicitement l'accès à une variable à l'aide d'une règle de réécriture $r_{\mathsf{var}} = \langle\mathsf{var}(x) \to x\rangle$. On définit ensuite une stratégie de réduction ϱ_2' qui diffère de ϱ_2 uniquement par l'ajout de la règle suivante :

$$\varrho_2'(\mathsf{var}(t)) \quad = \quad t$$

La dérivation de `phi` $(true, true)$ devient :

$$\begin{array}{rcl} \underline{\varphi}(true, true) & \to & \mathsf{if}(\underline{\mathsf{var}}(true), \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, true))) \\ & \to & \underline{\mathsf{if}}(true, \mathsf{true}, \varphi(\mathsf{true}, \varphi(\mathsf{false}, true))) \\ & \to & \underline{\mathsf{true}} \\ & \to & true. \end{array}$$

Elle a pour trace $\langle\!\langle r_\varphi, r_{\mathsf{var}}, r_{\mathsf{if}\,true}, r_{\mathsf{true}}\rangle\!\rangle$.

2.5.7 Schéma algébrique contre schéma régulier.

Il est difficile de représenter fidèlement les opérations liées à l'*environnement* parce qu'il est implicite. Il fait abstraitement partie du formalisme des schémas de programme, et n'est pas associé à un comportement dynamique. Cela offre à la fois des avantages et des inconvénients :

- Du point de vue simplement sémantique, cet environnement implicite fige la notion de *visibilité* (scope) des variables. Il correspond à la sémantique de la *liaison statique* (static binding) et ne permet pas de représenter la *liaision dynamique* (dynamic binding). Il ne permet pas non plus de décrire naturellement une construction de type « `let` ».

- Comme le montre l'exemple de l'accès explicite aux variables, l'emploi de cet environnement implicite compromet la fidélité du modèle d'exécution.

- Il n'y a pas de mécanisme pour manipuler une inconnue seule comme une valeur, un objet fonctionnel, par exemple pour le passer en paramètre, ni pour l'appliquer ensuite à des arguments.

- La formulation en termes de schémas algébriques permet l'étude générique de la correction des transformations de pliage/dépliage. Cependant, ces études s'appuyent principalement sur la sémantique $sem_{\boldsymbol{D}}$, qui reflète l'appel par nom, peu courant dans les langages de programmation ordinaires.

- D'autre part, la transformation de dépliage peut dupliquer des termes et ainsi modifier le comportement dynamique du programme résultant (cf. §1.3.4, objet des transformations). Ces copies, indépendantes de la nature de l'implémentation, sont indésirables.

C'est pourquoi nous préconisons l'emploi de schémas *réguliers* (les inconnues φ sont d'arité nulle) plutôt qu'*algébriques* (cf. §2.5.2, schéma de programme), qui explicitent une notion d'environnement.

- Un mécanisme explicite de composition des environnements détermine le type de visibilité des variables. Tout type de liaison est modélisable, ainsi que des constructions comme « `let` ».

- Les mécanismes d'appel fonctionnel et de sauvegarde dans l'environnement sont explicites. Ils permettent d'associer un comportement dynamique au stockage et l'accès à la valeur d'une variable.

- Les schémas réguliers n'ont pas de représentation spécialisée pour les fonctions. Une inconnue représente en quelque sorte un *pointeur* vers une valeur fonctionnelle, pointeur que l'on est libre de suivre ou non lors d'une exécution. Cette valeur fonctionnelle peut-être manipulée au même titre que les autres types de valeurs, puis appliquée à des paramètres. L'ajout d'opérateurs d'abstraction et d'application modélise explicitement les manipulations de *fermetures* (closures) des langages fonctionnels.

- Les schémas réguliers sont des cas particuliers de schémas algébriques. À ce titre, ils héritent des mêmes résultats, et notamment de ceux qui concernent la correction des transformations de pliage/dépliage. Ils ont même parfois des propriétés plus fortes parce qu'il n'y a pas d'expression emboîtée construite sur les inconnues, qui sont d'arité nulle. Puisque le mécanisme de passage de paramètres est explicite, ils peuvent *directement* modéliser des langages de programmation stricts, sans faire intervenir de stratégie de reduction. Ainsi, l'étude générique de la correction des transformations de programmes n'est pas limitée aux langages paresseux.

- À la différence des schémas algébriques, la transformation de dépliage ne crée pas de copie des arguments. Elle ne modifie donc pas le comportement dynamique des programmes.

Pour sa part, Courcelle [Cou86, §2.7] remarque que la théorie des schémas réguliers est plus simple et plus générale que celle des schémas algébriques, bien que transposer un résultat d'un formalisme à l'autre ne soit pas toujours immédiat. Nous indiquons dans les deux sections suivantes un moyen systématique de traduire une spécification qui emploie un schéma algébrique en une spécification à l'aide d'une schéma régulier.

2.5.8 Conversion d'un schéma algébrique en schéma régulier avec environnement implicite.

Courcelle donne dans [Cou86, §9] un procédé formel pour traduire un schéma de programme récursif algébrique en un schéma régulier, à l'aide d'opérateurs de composition et de projections. Si l'interprétation reste identique dans les deux formulations, aux quelques opérateurs supplémentaires près, la traduction nécessite toutefois une transformation importante des termes qui représentent les schémas. D'autre part, la visibilité des variables reste figée à la liaison statique et il n'est pas possible représenter simplement une construction comme « `let` ».

Procédé de conversion.

Nous proposons une autre traduction, qui préserve davantage la syntaxe des programmes au prix d'une modification de l'interprétation. Son intérêt est une représentation des variables plus naturelle et plus souple : on peut exprimer différents types de liaisons ; elle est aussi plus explicite, ce qui permet plus de fidélité dans une modélisation.

L'idée générale, très proche en fait des modélisations en *sémantique dénotationnelle* [Mos90], est la suivante :

- On considère les $x \in X_s$ comme des *noms de variables*, typés « $x : \mathrm{var}_s$ », et l'on adjoint à F des opérateurs $V = (\mathtt{var}_s)_{s \in \mathcal{S}}$, typés « $\mathtt{var}_s : \mathrm{var}_s \to s$ ». Les termes interprétés sont dans $M(F \cup V \cup X)$ et non dans $M(F)$. On parle de *variable sémantique* x pour les schémas algébriques (x a un rôle particulier) et de *variable syntaxique* dans le cas d'un schéma régulier (x et $\mathtt{var}_s$ sont intégrés à F, et manipulés comme autres opérateurs).
- On transforme les domaines d'interprétation D_s en des domaines $D'_s = \mathit{Env} \to D_s$ constitués de fonctions qui, appliquées à un environnement $E \in \mathit{Env} = X \to D$, retournent une valeur $d \in D_s$. Une

variable $\mathsf{var}_s(x)$ a pour interprétation la fonction[10] $\lambda(E{:}Env).E(x)$.

- On ajoute à F des opérateurs d'*appel* fonctionnel explicites « call » : à des indications de typage près, les termes $\varphi(t_1, \ldots, t_n)$ sont remplacés par $\mathsf{call}_n(\varphi, t_1, \ldots, t_n)$. On retrouve ainsi une syntaxe algébrique plus proche d'une syntaxe abstraite.
  Intuitivement, alors que le mécanisme de l'appel fonctionnel $\varphi(t_1, \ldots, t_n)$ dans les schémas algébriques est une construction purement mathématique, l'appel $\mathsf{call}_n(\varphi, t_1, \ldots, t_n)$ dans les schémas réguliers peut être compris comme le code d'un programme où φ désigne un pointeur, une adresse que l'on va simplement suivre pour exécuter la fonction correspondante. La transmission d'arguments est explicite et le modèle plus proche d'une implémentation réelle.
- Une définition $\varphi(x_1, \ldots, x_n) = t$ est récrite avec un opérateur d'abstraction, ou plus précisemment de *liaison de variables* : $\varphi = \mathsf{fun}_n(x_1, \ldots, x_n, t)$. Intuitivement, l'interprétation de $\mathsf{fun}_n(x_1, \ldots, x_n, t)$ est une fonction qui prend en argument des valeurs $(d_1, \ldots, d_n)$ de D et qui évalue le terme t dans l'environnement $E = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$.

Avant de donner une définition formelle de cette conversion, nous faisons une remarque qui vise à simplifier la spécification des schémas de programmes.

Des contraintes de typage font que les premiers arguments $x_1, \ldots, x_n$ de fun ne peuvent être que des *noms de variables* appartenant à X, et non des termes quelconques de $M(F' \cup V \cup X \cup \Phi)$. Non seulement une inconnue $\varphi \in \Phi$ ne peut pas apparaître sous ces occurrences, mais vouloir interpréter $\mathsf{fun}_n(\Omega, \ldots, \Omega, t)$ n'a pas beaucoup de sens ; dans ce modèle d'ailleurs, les noms de variables sont tous connus à tout moment de l'évaluation. Bien qu'il faille pourtant spécifier a priori une interprétation pour tous les opérateurs de la signature $F' \cup V \cup X$, l'ordre algébrique défini sur les noms de variables $x \in X$ ne joue en fait aucun rôle dans la sémantique : il est donc inutile de le mentionner. Cela se généralise à tout type de schéma, algébrique ou régulier, lorsqu'il existe une sorte $s \in \mathcal{S}$ telle que les inconnues $\varphi \in \Phi$ ne peuvent figurer dans les termes de type s. La proposition suivante formalise cette remarque.

Proposition 2.3. (Indépendance de certains ordres dans la sémantique des schémas de programme) Soient $\mathcal{S}$ un ensemble de sortes, F et Φ des $\mathcal{S}$-signatures, X une $\mathcal{S}$-signature de variables, et $\boldsymbol{D} = \langle (D_s)_{s\in\mathcal{S}}, (\leqslant_s)_{s\in\mathcal{S}}, (\perp_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{D}})_{f\in F}\rangle$ une F-interprétation. Alors pour tout F-schéma Σ d'inconnues Φ, $\Sigma_{\boldsymbol{D}}$ est indépendant de $(\leqslant_s)_{s\in\mathcal{S}'}$ et $(\perp_s)_{s\in\mathcal{S}'}$, avec $\mathcal{S}' = \{s \in \mathcal{S} \mid M_s(F \cup \Phi, X) = M_s(F, X)\}$.

Puisque $\Sigma_{\boldsymbol{D}}$ ne dépend pas des ordres sur $(D_s)_{s\in\mathcal{S}'}$, on peut les omettre dans une spécification et considérer implicitement que ces ordres sont, par exemple, des ordres plats. Cette proposition permet donc de plonger de la syntaxe algébrique dans une interprétation.

Démonstration. On a $\Sigma_{\boldsymbol{D}} = h_{\boldsymbol{D}}(\Sigma_{\boldsymbol{M}^\infty_\Omega(F,X)})$ (cf. §2.5.2) où $\Sigma_{\boldsymbol{M}^\infty_\Omega(F,X)}$ est l'arbre infini plus petite solution de Σ [Cou83], et $h_{\boldsymbol{D}}$ est l'homomorphisme d'algèbres continues de $\boldsymbol{M}^\infty_\Omega(F,X)$ dans $\boldsymbol{D}$. Les sous-termes t de $\Sigma_{\boldsymbol{M}^\infty_\Omega(F,X)}$ de sortes $s \in \mathcal{S}'$ sont finis puisque $M_s(F \cup \Phi, X) = M_s(F, X)$. L'ordre $(\leqslant_s, \perp_s)$ sur D_s, et notamment l'existence d'un « sup », n'intervient donc pas dans le calcul de $t_{\boldsymbol{D}} = h_{\boldsymbol{D}}(t)$, et par conséquent pas non plus dans celui de $(\Sigma_{\boldsymbol{M}^\infty_\Omega(F,X)})_{\boldsymbol{D}}$. $\square$

Il est donc inutile de donner une structure ordonnée aux noms de variables de X. C'est pour cela d'ailleurs que nous aurions pu employer une famille d'opérateurs $(\mathsf{fun}_{x_1\ldots x_n})_{x_1,\ldots,x_n\in X}$ d'arité 1, plutôt que des opérateurs

[10] Rappelons que dans ce document, « λ » permet simplement une *notation* concise des fonctions, et n'a pas d'autre rapport avec le λ-calcul.

fun_n d'arité $n+1$. À présent, la définition suivante spécifie plus formellement comment convertir un schéma algébrique en schéma régulier.

Définition 2.9. (Procédé de conversion en schéma régulier avec environnement implicite)
Soient $\mathcal{S}$ un ensemble de sortes, F une $\mathcal{S}$-signature, X une $\mathcal{S}$-signature de variables, Σ un F-schéma *algébrique* sur les inconnues Φ, et $\boldsymbol{D} = \langle (D_s)_{s\in\mathcal{S}}, (\leqslant_s)_{s\in\mathcal{S}}, (\perp_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{D}})_{f\in F}\rangle$ une F-interprétation. La *conversion en schéma régulier* de la spécification $(\Sigma, \boldsymbol{D})$ est la spécification $(\Sigma', \boldsymbol{D}') = Reg(\Sigma, \boldsymbol{D})$, définie par $\mathcal{S}'$ un ensemble de sortes, F' une $\mathcal{S}'$-signature, Σ' un F'-schéma *régulier* sur les inconnues Φ', et $\boldsymbol{D}' = \langle (D'_s)_{s\in\mathcal{S}'}, (\leqslant'_s)_{s\in\mathcal{S}'}, (\perp'_s)_{s\in\mathcal{S}'}, (f'_{\boldsymbol{D}'})_{f'\in F'}\rangle$ une F'-interprétation tels que :

- $\mathcal{S}' = \mathcal{S} \cup \{\mathrm{var}_s\ ,\ \mathrm{fun}_{s_1\ldots s_n s} \mid s, s_1, \ldots, s_n \in \mathcal{S}\}$
- $X'_s = \{x : \mathrm{var}_s \mid x \in X_s\}$ pour $s \in \mathcal{S}$, et $X' = (X'_s)_{s\in\mathcal{S}}$
- $F' = F \cup X' \cup \{\mathsf{call}_{s_1\ldots s_n s} : \mathrm{fun}_{s_1\ldots s_n s} \times s_1 \times \cdots \times s_n \to s\ ,\ \mathsf{fun}_{s_1\ldots s_n s} : \mathrm{var}_{s_1} \times \cdots \times \mathrm{var}_{s_n} \times s \to \mathrm{fun}_{s_1\ldots s_n s}\ ,\ \mathsf{var}_s : \mathrm{var}_s \to s \mid s, s_1, \ldots, s_n \in \mathcal{S}\}$
- $\Phi' = \{\varphi : \mathrm{fun}_{s_1\ldots s_n s} \mid \langle \varphi : s_1 \times \cdots \times s_n \to s\rangle \in \Phi\}$
- Σ' est déduit de Σ en remplaçant, pour tout $\langle \varphi : s_1 \times \cdots \times s_n \to s\rangle \in \Phi$,
  - la définition $\langle \varphi(x_1, \ldots, x_n) = t\rangle$ par $\langle \varphi = \mathsf{fun}_{s_1\ldots s_n s}(x_1, \ldots, x_n, t)\rangle$,
  - les termes « $\varphi(t_1, \ldots, t_n)$ » par « $\mathsf{call}_{s_1\ldots s_n s}(\varphi, t_1, \ldots, t_n)$ » .

Et pour tout $s, s_1, \ldots s_n \in \mathcal{S}$,

- $Env = X' \overset{\text{fin}}{\to} D$ tq $\forall E \in Env \quad \forall x \in X'_s \cap Dom(E) \quad E(x) \in D_s$
- $D'_s = Env \to D_s$
- $\perp'_s = \lambda E.\perp_s$
- $d'_1 \leqslant'_s d'_2$ ssi $d'_1(E) \leqslant_s d'_2(E)$ pour tous $d'_1, d'_2 \in D'_s$ et $E \in Env$
- $D'_{\mathrm{var}_s} = X'_s$, muni d'un ordre plat (cf. prop. 2.3).
- $D'_{\mathrm{fun}_{s_1\ldots s_n s}} = D'_s \to (D_{s_1} \times \cdots \times D_{s_n} \to D_s)$
- $\perp'_{\mathrm{fun}_{s_1\ldots s_n s}} = \lambda d'.\lambda(d_1, \ldots, d_n).\perp_s$
- $g_1 \leqslant'_{\mathrm{fun}_{s_1\ldots s_n s}} g_2$ ssi $g_1(d')(d_1, \ldots, d_n) \leqslant_s g_2(d')(d_1, \ldots, d_n)$ pour tout $(d_1, \ldots, d_n) \in D_{s_1} \times \cdots \times D_{s_n}$, pour tout $d' \in D'_s$ et pour tous $g_1, g_2 \in D'_{\mathrm{fun}_{s_1\ldots s_n s}}$
- $\forall f \in F_n \quad f_{\boldsymbol{D}'} = \lambda(d'_1, \ldots, d'_n).\lambda E.f_{\boldsymbol{D}}(d'_1(E), \ldots, d'_n(E))$
- $\forall x \in X_s \quad x_{\boldsymbol{D}'} = x$
- $\mathsf{var}_{s\boldsymbol{D}'} = \lambda x.\lambda E.\,\text{si } x \in Dom(E) \text{ alors } E(x) \text{ sinon } \perp_s$
- $\mathsf{call}_{s_1\ldots s_n s\boldsymbol{D}'} = \lambda(g, d'_1, \ldots d'_n).\lambda E.g(d'_1(E), \ldots, d'_n(E))$
- $\mathsf{fun}_{s_1\ldots s_n s\boldsymbol{D}'} = \lambda(x_1, \ldots, x_n, d').\lambda(d_1, \ldots, d_n).\, d'(\{x_i \mapsto d_i \mid i \in [n],\ x_i \neq \perp_{\mathrm{var}_{s_i}}\})$

On peut étendre la F'-interprétation $\boldsymbol{D}'$ en une $(F' \cup D)$-interprétation par $d_{\boldsymbol{D}'} = \lambda E.d$ pour $d \in D_s$.

Parce que les fonctions d'une algèbre sont par définition des fonctions *totales*, on a défini l'interprétation d'une variable $\mathsf{var}_s(x)$ comme « $\lambda E.\,\text{si } x \in Dom(E) \text{ alors } E(x) \text{ sinon } \perp_s$ » et non « $\lambda E.E(x)$ ». Confondre la valeur représentant une erreur (si l'on accède à une variable indéfinie) avec l'élément $\perp$ (censé représenter la non-terminaison) est en général un amalgame peu élégant. Ce n'est pas important ici car la condition $Var(t) \subset \{x_1, \ldots, x_n\}$ pour tout $\langle \varphi(x_1, \ldots x_n) = t\rangle \in \Sigma$ est préservée par la traduction (sous une forme syntaxique différente), et les variables à interpréter sont donc toujours définies dans l'environnement courant. La même remarque explique l'interprétation de l'opérateur $\mathsf{fun}_{s_1\ldots s_n s}$.

Cette définition ($\boldsymbol{D}'$ est bien une F'-algèbre continue, cf. §D.3) trouve son intérêt dans la proposition suivante.

Proposition 2.4. (Correction de la transformation en schéma régulier avec environnement implicite) Soient $(\Sigma, \boldsymbol{D})$ une F-spécification et $(\Sigma', \boldsymbol{D}') = Reg(\Sigma, \boldsymbol{D})$. Alors $\Sigma_{\boldsymbol{D}} = \Sigma'_{\boldsymbol{D}'}$.

Démonstration. Reprenons les notations adoptées pour la sémantique du plus petit point fixe (cf. §2.5.2) et annotons d'un « $'$ » tous les objets relatifs à Σ' et $\boldsymbol{D}'$. On démontre que $G = G'$, c'est-à-dire que pour tout $i \in [N]$, $\hat{g}_i = \hat{g}'_i$. Pour cela, on montre par récurrence structurelle sur t_i que $(t_i)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]} = (t'_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]}$. Or la traduction est telle que $t'_i = \mathsf{fun}_{\sigma(x_{i,1})\ldots\sigma(x_{i,n_i})\sigma(\varphi_i)}(x_{i,1}, \ldots, x_{i,n_i}, t''_i)$ avec $t''_i \in M(F' \cup \Phi')$, donc pour tout $(d_1, \ldots, d_{n_i}) \in D_i$ $(t'_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = (t''_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(E_i(d_1, \ldots, d_{n_i}))$ en notant $E_i(d_1, \ldots, d_{n_i}) = \{x_{i,1} \mapsto d_1, \ldots, x_{i,n_i} \mapsto d_{n_i}\} \in Env$. Il y a trois cas :

- Si $t_i = x_{i,j}$ avec $j \in [n_i]$ alors $t''_i = \mathsf{var}_{\sigma(x_{i,j})}(x_{i,j})$. On a $(t_i)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = d_j$ et $(t''_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]} = \lambda E.\, \text{si } x_{i,j} \in Dom(E) \text{ alors } E(x_{i,j}) \text{ sinon } \perp_{\sigma(x_{i,j})}$, donc $(t''_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(E_i(d_1, \ldots, d_{n_i})) = d_j = (t_i)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i})$.
- Si $t_i = f(u_1, \ldots, u_m)$ avec $f \in F_m$ alors t''_i s'écrit $f(u'_1, \ldots, u'_m)$. Supposons que $(u_j)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = (u'_j)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(E_i(d_1, \ldots, d_{n_i}))$ pour tout $j \in [m]$. Alors $(t_i)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = f_D(\ldots, (u_j)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}), \ldots)$ et $(t''_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]} = \lambda E. f_{\boldsymbol{D}}(\ldots, (u'_j)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(E), \ldots)$, donc $(t_i)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = (t''_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(E_i(d_1, \ldots, d_{n_i}))$.
- Si $t_i = \varphi_k(u_1, \ldots, u_{n_k})$ avec $\varphi_k \in \Phi$ alors t''_i s'écrit $\mathsf{call}_{\sigma(x_{k,1})\ldots\sigma(x_{k,n_k})\sigma(\varphi_k)}(\varphi_k, u'_1, \ldots, u'_{n_k})$. Supposons que $(u_j)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = (u'_j)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(E_i(d_1, \ldots, d_{n_i}))$ pour tout $j \in [n_k]$. Alors $(t_i)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = g_k(\ldots, (u_j)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}), \ldots)$ et $(t''_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]} = \lambda E. g_k(\ldots, (u'_j)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}), \ldots)$, donc $(t_i)_{X_i, \boldsymbol{D}[g_1,\ldots,g_N]}(d_1, \ldots, d_{n_i}) = (t''_i)_{\boldsymbol{D}'[g_1,\ldots,g_N]}(E_i(d_1, \ldots, d_{n_i}))$.

Par conséquent, $\Sigma_{\boldsymbol{D}} = \text{lfp}(G) = \text{lfp}(G') = \Sigma'_{\boldsymbol{D}'}$, car $\widetilde{\Delta} = \widetilde{\Delta}'$ sont munis du même ordre. □

Exemple de conversion.

Syntaxe algébrique. La syntaxe algébrique en termes de schémas réguliers coïncide avec la syntaxe abstraite de BOOL, à l'exception du mécanisme de définition de fonctions qui reste exprimé sous forme d'équations.

- $\mathcal{S}' = \{\text{bool}, \text{var}, \text{fun}_n \mid n \in \mathbb{N}\}$
- $F' = \{\mathsf{true} : \text{bool}\ ,\ \mathsf{false} : \text{bool}\ ,\ \mathsf{var} : \text{var} \to \text{bool}\ ,\ \mathsf{if} : \text{bool} \times \text{bool} \times \text{bool} \to \text{bool}\ ,\ \mathsf{call}_n : \text{fun}_n \times \text{bool}^n \to \text{bool}\ ,\ \mathsf{fun}_n : \text{var}^n \times \text{bool} \to \text{fun}_n \mid n \in \mathbb{N}\}$
- $\Phi' \subset \{\varphi_{fun} : \text{fun}_n \mid fun \in \text{Fun},\ n \in \mathbb{N}\}$

Parce que BOOL n'a essentiellement qu'un type, l'annotation par $s_1 \ldots s_n s$ est réduite à l'arité n.

Traduction dans les schémas. Une traduction $T : \text{Dec} \to Sch_{reg}(F')$ est donnée à la figure 2.12.

La fonction `nand` définie précédemment (cf. §2.2.2) avait pour schéma algébrique « $\textbf{nand}(x, y) = \mathsf{if}(x, \mathsf{if}(y, \mathsf{false}, \mathsf{true}), \mathsf{true})$ ». Sa formulation en termes de schéma régulier est :

$$\textbf{nand} = \mathsf{fun}_2(x, y, \mathsf{if}(\mathsf{var}(x), \mathsf{if}(\mathsf{var}(y), \mathsf{false}, \mathsf{true}), \mathsf{true}))$$

De même, l'appel $\textbf{nand}(\mathsf{true}, \mathsf{false})$ est codé $\mathsf{call}_2(\textbf{nand}, \mathsf{true}, \mathsf{false})$.

Avec ce type formulation il est possible de représenter un terme comme $\mathsf{fun}(x, \mathsf{var}(y))$, traduction du programme éronné « `fun f(X) = Y` », alors qu'une condition syntaxique en interdisait la construction avec

$$
\begin{array}{lcl}
T(\texttt{true}) & = & \mathsf{true} \\
T(\texttt{false}) & = & \mathsf{false} \\
T(var) & = & \mathsf{var}(x_{var}) \\
T(\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3) & = & \mathsf{if}(T(exp_1), T(exp_2), T(exp_3)) \\
T(fun\texttt{(}exp_1\texttt{, \ldots ,}exp_n\texttt{)}) & = & \mathsf{call}_n(fun, T(exp_1), \ldots, T(exp_n)) \\
 & & \\
T(fun\texttt{(}var_1\texttt{, \ldots ,}var_n\texttt{) = }exp) & = & \varphi_{fun} = \mathsf{fun}_n(x_{var_1}, \ldots, x_{var_n}, T(exp)) \\
T(funbind\ \langle\texttt{and } funbinds\rangle) & = & \{T(funbind)\}\ \langle \cup\, T(funbinds)\rangle \\
T(\texttt{fun } funbinds) & = & T(funbinds)
\end{array}
$$

Figure 2.12 : Traduction des programmes BOOL dans les schémas réguliers

des schémas algébriques (cf. §2.5.2, schéma de programme). C'est la rançon d'une formalisation plus générale des environnements. Toutefois, le problème ne se pose pas pour BOOL car nous avons donné la même condition syntaxique (cf. §2.2.1) afin précisément de pas avoir à traiter la production d'une erreur à l'accès d'une variable.

Interprétation et sémantique. Soit $\boldsymbol{D}' = \langle (D'_s, \leqslant'_s, \bot'_s)_{s\in S'}, (f_{\boldsymbol{D}'})_{f\in F'}\rangle$ la F'-interprétation suivante.

- $\mathrm{VarEnv} = \mathrm{Var} \stackrel{\mathrm{fin}}{\to} \mathrm{BoolVal}_\bot$
- $D'_{\mathrm{bool}'} = \mathrm{VarEnv} \to \mathrm{BoolVal}_\bot$ et $\bot_{\mathrm{bool}'} = \lambda E.\bot$
- $D'_{\mathrm{fun}_n} = \mathrm{BoolVal}^n_\bot \to \mathrm{BoolVal}_\bot$ et $\bot_{\mathrm{fun}_n} = \lambda v_1 \ldots v_n.\bot$
- $D'_{\mathrm{var}} = \mathrm{Var}$, et les ordres $(\leqslant_s)_{s\in S'}$ sont tels que $\boldsymbol{D}$ est une interprétation discrète.
- $\mathsf{true}_{\boldsymbol{D}'} = \lambda E.true$
- $\mathsf{false}_{\boldsymbol{D}'} = \lambda E.false$
- $\mathsf{if}_{\boldsymbol{D}'}(b_1, b_2, b_3) = \lambda E. \begin{cases} b_2(E) & \text{si } b_1(E) = true \\ b_3(E) & \text{si } b_1(E) = false \\ \bot & \text{si } b_1(E) = \bot \end{cases}$
- $\mathsf{var}_{\boldsymbol{D}'}(x) = \lambda E.\,\text{si } x \in Dom(E) \text{ alors } E(x) \text{ sinon } \bot$
- $\mathsf{call}_{n\boldsymbol{D}'}(f, b_1, \ldots, b_n) = \lambda E.\, f(b_1(E), \ldots, b_n(E))$
- $\mathsf{fun}_{n\boldsymbol{D}'}(x_1, \ldots, x_n, b) = \lambda v_1 \ldots v_n. b(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\})$

On retrouve bien $\mathbf{nand}_{\boldsymbol{D}'}(true, false) = true$. Notons aussi qu'il est désormais possible de forcer la sémantique de l'appel par valeur directement au niveau de l'interprétation en définissant :

- $\mathsf{call}_{n\boldsymbol{D}'}(f, b_1, \ldots, b_n) = \lambda E.\,\text{si (pour tout } i \in [n] \quad b_i(E) \neq \bot) \text{ alors } f(b_1(E), \ldots, b_n(E)) \text{ sinon } \bot$

Ce n'est pas spécifique à BOOL mais général à tout schéma régulier obtenu par le procédé de la définition 2.9.

Réécritures sémantiques.

Cette formulation pose toutefois un problème pour transporter ou pour définir des stratégies de réduction car les domaines sont plus complexes : les valeurs manipulées dans la sémantique opérationnelle sont des objets fonctionnels.

Alors qu'un schéma algébrique Σ permet de dériver un *appel fonctionnel* $\varphi(d_1, \ldots, d_n) \to^* d \in D$, le schéma régulier Σ' correspondant ne peut que déterminer la *fonction* $\varphi_{\boldsymbol{D}'}$, et ce comme la borne supérieure d'ensembles de dérivations, pour l'appliquer ensuite à $(d_1, \ldots, d_n)$. Dans le cas très particulier où $\boldsymbol{D}$ est constitué de domaines finis, comme c'est le cas pour BOOL, il n'existe qu'un nombre fini de fonctions typées $\mathrm{fun}_{s_1 \ldots s_n s}$ et l'interprétation $\boldsymbol{D}'$ est donc également constituée de domaines finis ; la fonction $\varphi_{\boldsymbol{D}'}$ peut alors être approximée et atteinte par une dérivation finie.

Mais dans le cas général, $\boldsymbol{D}'$ contient des domaines infinis et il existe des fonctions $\varphi_{\boldsymbol{D}'}$ qui ne sont pas *effectivement* calculables car elles peuvent être approchées en un nombre de pas infinis. Ce n'est pas gênant en pratique car on ne cherche pas à calculer $\varphi_{\boldsymbol{D}'}$ mais l'application de $\varphi_{\boldsymbol{D}'}$ à des données $(d_1, \ldots, d_n)$ de $\boldsymbol{D}$.

Remarquons toutefois que la sémantique de l'opérateur call se traduit par :

$$\lambda E.\varphi_{\boldsymbol{D}'}(d_1, \ldots, d_n) = (\mathsf{call}_{s_1 \ldots s_n s}(\varphi, \lambda E.d_1, \ldots, \lambda E.d_n))_{\boldsymbol{D}'}$$

Avec cette formulation, il est possible de faire la dérivation suivante de $\mathbf{fac}(3)$, au moyen des règles $R(\boldsymbol{D}')$ (cf. §2.5.2, sémantique opérationnelle), dans une interprétation $\boldsymbol{D}'$ que nous n'explicitons pas (cf. §2.5.1) :

$$\begin{array}{lll}
\mathsf{call}(\underline{\mathbf{fac}}, \lambda E.3) & \overset{4}{\underset{\Sigma}{\to}} \underset{\Omega}{\to} & \mathsf{call}(\underline{\mathsf{fun}}(n, \mathsf{ifz}(\mathsf{var}(n), 1, \mathsf{mul}(\mathsf{var}(n), \mathsf{call}(\\
& & \quad \mathsf{fun}(n, \mathsf{ifz}(\mathsf{var}(n), 1, \mathsf{mul}(\mathsf{var}(n), \mathsf{call}(\\
& & \quad\; \mathsf{fun}(n, \mathsf{ifz}(\mathsf{var}(n), 1, \mathsf{mul}(\mathsf{var}(n), \mathsf{call}(\\
& & \quad\;\; \mathsf{fun}(n, \mathsf{ifz}(\mathsf{var}(n), 1, \mathsf{mul}(\mathsf{var}(n), \mathsf{call}(\,\Omega\,, \ldots \mathsf{sub1}(\mathsf{var}(n))))))))))), \lambda E.3) \\
& \overset{*}{\underset{R(\boldsymbol{D})}{\longrightarrow}} & \underline{\mathsf{call}}(\lambda n.\ \text{si } n = 0 \ \text{ alors } 1 \ \text{sinon} \\
& & \qquad \text{si } n = 1 \ \text{ alors } 1 \ \text{sinon} \\
& & \qquad\; \text{si } n = 2 \ \text{ alors } 2 \ \text{sinon} \\
& & \qquad\;\; \text{si } n = 3 \ \text{ alors } 6 \ \text{sinon } \bot,\ \lambda E.3) \\
& \underset{R(\boldsymbol{D})}{\longrightarrow} & \lambda E.6\,.
\end{array}$$

Non seulement, il faut « deviner » le nombre de dépliages de **fac** nécessaires en fonction de l'argument de call (ici au moins quatre dépliages car l'argument est $\lambda E.3$), mais c'est également un piètre modèle d'exécution car toutes les valeurs intermédiaires ($fac(0) = 1,\ fac(1) = 1,\ fac(2) = 2$) doivent nécessairement être calculées. En effet, les règles de $R(\boldsymbol{D}')$ sont de la forme $\langle f(t_1, \ldots, t_n) \to t_0 \rangle$ où $t_i \in D \cup X$ et telles que $(f(t_1, \ldots, t_n))_{\boldsymbol{D}} = (t_0)_{\boldsymbol{D}}$. Pour effectuer la dernière réduction d'une dérivation il faut donc disposer d'un approximant fac de $\mathbf{fac}_{\boldsymbol{D}'}$ afin de conclure par la règle $\langle \mathsf{call}(fac, d'_1, \ldots, d'_n) \to \lambda E.fac(d'_1(E), \ldots, d'_n(E)) \rangle \in R(\boldsymbol{D}')$. On voit dans l'exemple ci-dessus que cet appromixant est inutilement précis.

C'est pourquoi l'on considère les régles plus générales de $V(\boldsymbol{D}')$, de la forme $\langle t \to t' \rangle$ avec $t, t' \in M(F, X)$ et $t_{\boldsymbol{D}'} = t'_{\boldsymbol{D}'}$. On dérive ainsi :

$$\begin{array}{lll}
\mathsf{call}(\underline{\mathbf{fac}}, \lambda E.3) & \underset{\Sigma}{\longrightarrow} & \underline{\mathsf{call}}(\mathsf{fun}(n, \mathsf{ifz}(\mathsf{var}(n), 1, \mathsf{mul}(\mathsf{var}(n), \mathsf{call}(\mathbf{fac}, \mathsf{sub1}(\mathsf{var}(n))))))), \lambda E.3) \\
& \underset{V(\boldsymbol{D}')}{\longrightarrow} & \mathsf{call}(\mathsf{fun}(n, \mathsf{mul}(\mathsf{var}(n), \mathsf{call}(\underline{\mathbf{fac}}, \mathsf{sub1}(\mathsf{var}(n)))))), \lambda E.3) \\
& \overset{*}{\underset{\Sigma, V(\boldsymbol{D}')}{\longrightarrow}} & \underline{\mathsf{call}}(\mathsf{fun}(n, \mathsf{mul}(\mathsf{var}(n), \\
& & \quad \mathsf{call}(\mathsf{fun}(n, \mathsf{mul}(\mathsf{var}(n), \\
& & \quad\; \mathsf{call}(\mathsf{fun}(n, \mathsf{mul}(\mathsf{var}(n), \\
& & \quad\;\; \mathsf{call}(\mathsf{fun}(n, 1), \mathsf{sub1}(\mathsf{var}(n))))), \mathsf{sub1}(\mathsf{var}(n))))), \mathsf{sub1}(\mathsf{var}(n))))), \lambda E.3) \\
& \overset{*}{\underset{V(\boldsymbol{D}')}{\longrightarrow}} & \lambda E.6\,.
\end{array}$$

Cette dérivation insolite n'effectue en apparence que des calculs nécessaires. En particulier, elle ne calcule pas, directement ou indirectement, les valeurs $fac(0)$, $fac(1)$, $fac(2)$. Mais elle est néanmoins un peu artificielle car, de même qu'il fallait « deviner » le nombre de dépliages de **fac** dans l'exemple précédent, il faut ici aussi propager implicitement la valeur $\lambda E.3$ dans tout le terme afin de savoir quelles réductions appliquer pour « évaluer » les constructions if.

2.5.9 Conversion d'un schéma algébrique en schéma régulier avec environnement explicite.

Pour éliminer la facticité des réductions de l'exemple précédent, pour les rendre plus naturelles et plus à même de modéliser une implémentation, nous explicitons la notion d'environnement au niveau syntaxique et non pas uniquement sémantique.

Procédé de conversion.

Nous introduisons pour cela une nouvelle sorte env, ainsi que des opérateurs « $\mathsf{in}_s : \mathrm{env} \times s \to s$ ». Une interprétation $\boldsymbol{D}''$ sur la signature F'' correspondante fait en sorte que l'on puisse réduire par exemple $\mathsf{in}(\{x \mapsto v\}, \mathsf{var}(x)) \to \lambda E.v$. Considérer $\boldsymbol{D}''$ comme une $(F'' \cup D)$-interprétation, comme nous l'avions fait à la définition 2.9 pour $\boldsymbol{D}'$, et l'ensemble de règles $V(\boldsymbol{D}'')$ plutôt que $R(\boldsymbol{D}'')$ (cf. déf. 2.5, prop. 2.1), permet en outre d'« abréger » $\lambda E.v$ en v.

Définition 2.10. (Procédé de conversion en schéma régulier avec environnement explicite)
Soient $(\Sigma, \boldsymbol{D})$ une F-spécification et $(\Sigma', \boldsymbol{D}') = Reg(\Sigma, \boldsymbol{D})$. La F''-spécification *régulière* $Reg_{\mathrm{env}}(\Sigma, \boldsymbol{D}) = (\Sigma'', \boldsymbol{D}'')$ est définie par $\Sigma'' = \Sigma'$ et $\boldsymbol{D}''$, *extension* (cf. §N.13) de $\boldsymbol{D}'$ telle que :

- $\mathcal{S}'' = \mathcal{S}' \cup \{\mathrm{env}\}$
- $F'' = F' \cup \{\mathsf{in}_s : \mathrm{env} \times s \to s \mid s \in \mathcal{S}\} \cup \{E : \mathrm{env} \mid E \in Env\}$
- $\boldsymbol{D}''_{\mathrm{env}} = Env$, muni d'un ordre plat (cf. prop. 2.3).
- $\mathsf{in}_{s\boldsymbol{D}''} = \lambda(E_1, d'').\lambda E_2.d''(E_1)$

On peut étendre la F''-interprétation $\boldsymbol{D}''$ en une $(F'' \cup D)$-interprétation par $d_{\boldsymbol{D}''} = \lambda E.d$ pour tout $d \in D$.

Nous avons fait apparaître explicitement les types (qui n'apportent pas grand chose en plus) afin de ne pas avoir à spécifier quels sont les termes bien formés d'un langage. Outre une équivalence similaire à celle de la proposition 2.4, on dispose d'identités qui facilitent la modélisation d'implémentations courantes :

Proposition 2.5. (Correction de la transformation en schéma régulier avec environnement explicite)
Soient $(\Sigma, \boldsymbol{D})$ une F-spécification et $(\Sigma'', \boldsymbol{D}'') = Reg_{\mathrm{env}}(\Sigma, \boldsymbol{D})$. Alors :

- $\Sigma_{\boldsymbol{D}} = \Sigma''_{\boldsymbol{D}''}$

Avec les notations ci-dessus, on a de plus les identités suivantes pour $\langle \varphi(x_1, \ldots, x_n) = t \rangle \in \Sigma$:

- $\mathsf{call}_{s_1 \ldots s_n s}(\mathsf{fun}_{s_1 \ldots s_n s}(x_1 \ldots x_n, t), t_1 \ldots t_n)_{\boldsymbol{D}''}(E) = \mathsf{in}_s(\{x_1 \mapsto t_{1\boldsymbol{D}''}(E) \ldots x_n \mapsto t_{n\boldsymbol{D}''}(E)\}, t)_{\boldsymbol{D}''}(E_0)$
- $\lambda E.\varphi_{\boldsymbol{D}}(d_1, \ldots, d_n) = \mathsf{in}(\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}, t)_{\boldsymbol{D}''}$

où E_0 est un environnement quelconque, par exemple vide.

Démonstration. Il est clair que $\Sigma'_{\boldsymbol{D}'} = \Sigma''_{\boldsymbol{D}'} = \Sigma''_{\boldsymbol{D}''}$, car l'interprétation $\boldsymbol{D}''$ est une extension (cf. §N.13) de $\boldsymbol{D}'$. Or $\Sigma_{\boldsymbol{D}} = \Sigma'_{\boldsymbol{D}'}$ (prop. 2.4), donc $\Sigma_{\boldsymbol{D}} = \Sigma''_{\boldsymbol{D}''}$.

- $\mathsf{call}_{s_1\ldots s_n s}(\mathsf{fun}_{s_1\ldots s_n s}(x_1,\ldots,x_n,t),t_1,\ldots,t_n)_{\boldsymbol{D}''}(E) = \mathsf{call}_{s_1\ldots s_n s\boldsymbol{D}''}(\lambda(d_1\ldots,d_n).t_{\boldsymbol{D}''}(\{x_1 \mapsto d_1,\ldots, x_n \mapsto d_n\}), t_{1\boldsymbol{D}''},\ldots,t_{n\boldsymbol{D}''})(E) = t_{\boldsymbol{D}''}(\{x_1 \mapsto t_{1\boldsymbol{D}''}(E),\ldots,x_n \mapsto t_{n\boldsymbol{D}''}(E)\}) = \mathsf{in}_{s\boldsymbol{D}''}(\{x_1 \mapsto t_{1\boldsymbol{D}''}(E),\ldots,x_n \mapsto t_{n\boldsymbol{D}''}(E)\}, t_{\boldsymbol{D}''})(E_0)$ avec E_0 quelconque.
- $\lambda E.\varphi_{\boldsymbol{D}}(d_1,\ldots,d_n) = \mathsf{call}_{s_1\ldots s_n s}(\varphi, \lambda E.d_1,\ldots,\lambda E.d_n)_{\boldsymbol{D}''} = \lambda E.(\mathsf{call}_{s_1\ldots s_n s}(\mathsf{fun}_{s_1\ldots s_n s}(x_1,\ldots,x_n,t), \lambda E.d_1,\ldots,\lambda E.d_n)_{\boldsymbol{D}''}(E)) = \lambda E.(\mathsf{in}(\{x_1 \mapsto d_1,\ldots,x_n \mapsto d_n\},t)_{\boldsymbol{D}''}(E_0)) = \mathsf{in}(\{x_1 \mapsto d_1,\ldots,x_n \mapsto d_n\},t)_{\boldsymbol{D}''}$, en utilisant le point précédent. □

Cette formulation permet de modéliser facilement des types de visibilité différents. Par exemple, on peut donner la sémantique suivante à in_s :

- $\mathsf{in}_{s\boldsymbol{D}''} = \lambda(E_2, d'').\lambda E_1.d''(E_1 + E_2)$

où la fonction $+ : \mathit{Env} \times \mathit{Env} \to \mathit{Env}$ est une loi de composition des environnements. Par exemple, pour tous $E_1, E_2 \in \mathit{Env}$ et $x \in \mathit{Dom}(E_1 + E_2)$,

- $\mathit{Dom}(E_1 + E_2) = \mathit{Dom}(E_1) \cup \mathit{Dom}(E_2)$
- $(E_1 + E_2)(x) = \text{si } x \in \mathit{Dom}(E_2) \text{ alors } E_2(x) \text{ sinon } E_1(x)$

Non seulement on spécifie ainsi la *sémantique* de la *liaison dynamique*, mais il est aussi possible de décrire un *comportement dynamique* qui modélise une implémentation.

Exemple de conversion.

Spécification. Dans la cas de BOOL, on a donc :

- $\mathcal{S}'' = \mathcal{S}' \cup \{\mathrm{env}\} = \{\mathrm{bool}, \mathrm{var}, \mathrm{env}, \mathrm{fun}_n \mid n \in \mathbb{N}\}$
- $F'' = F' \cup \{\mathsf{in} : \mathrm{env} \times \mathrm{bool} \to \mathrm{bool}\} \cup \{E : \mathrm{env} \mid E \in \mathrm{VarEnv}\}$

Nous considérons aussi $\boldsymbol{D}''$ comme la $(F'' \cup \mathrm{BoolVal})$-interprétation décrite à la définition 2.10.

Règles de réécritures. La figure 2.13 donne quelques règles de $V^\dagger(\boldsymbol{D}'')$. Notons que les règles $r_{\mathsf{in\ if}}$ et $r_{\mathsf{in\ call}_n}$ ne sont ni des règles de $R(\boldsymbol{D}'')$, ni des règles de simplification [RV80] car la taille de leur partie droite n'est pas plus petite que celle de leur partie gauche. Cela justifie a posteriori l'introduction des règles $V(\boldsymbol{D})$ (déf. 2.5).

$$
\begin{array}{rcll}
\mathsf{if}(\mathit{true}, t_2, t_3) & \to & t_2 & (r_{\mathsf{if}\,\mathit{true}}) \\
\mathsf{if}(\mathit{false}, t_2, t_3) & \to & t_3 & (r_{\mathsf{if}\,\mathit{false}}) \\
\mathsf{call}_n(\mathsf{fun}_n(x_1,\ldots,x_n,t), d_1,\ldots,d_n) & \to & \mathsf{in}(\{x_1 \mapsto d_1,\ldots,x_n \mapsto d_n\}, t) & (r_{\mathsf{call}_n}) \\
\mathsf{in}(E, \mathsf{true}) & \to & \mathit{true} & (r_{\mathsf{true}}) \\
\mathsf{in}(E, \mathsf{false}) & \to & \mathit{false} & (r_{\mathsf{false}}) \\
\mathsf{in}(E, \mathsf{var}(x)) & \to & E(x) & (r_{\mathsf{var}}) \\
\mathsf{in}(E, \mathsf{if}(t_1,t_2,t_3)) & \to & \mathsf{if}(\mathsf{in}(E,t_1), \mathsf{in}(E,t_2), \mathsf{in}(E,t_3)) & (r_{\mathsf{in\ if}}) \\
\mathsf{in}(E, \mathsf{call}_n(f, t_1,\ldots,t_n)) & \to & \mathsf{call}_n(f, \mathsf{in}(E,t_1),\ldots,\mathsf{in}(E,t_n)) & (r_{\mathsf{in\ call}_n})
\end{array}
$$

Figure 2.13 : Quelques règles de $V^\dagger(\boldsymbol{D}'')$, avec $\boldsymbol{D}''$ considéré comme une $(F'' \cup D)$-interprétation

Stratégie de réduction. La figure 2.14 donne un exemple de stratégie de réduction pour BOOL qui utilise ces règles. Elle permet de construire par exemple la dérivation suivante, où l'on a posé $E = \{x \mapsto \mathit{true}, y \mapsto \mathit{false}\}$:

$$
\begin{array}{lcl}
\mathsf{call}_2(\underline{\mathbf{nand}}, \mathit{true}, \mathit{false}) & \rightarrow & \underline{\mathsf{call}_2(\mathsf{fun}_2}(x,y,\mathsf{if}(\mathsf{var}(x),\mathsf{if}(\mathsf{var}(y),\mathsf{false},\mathsf{true}),\mathsf{true})),\mathit{true},\mathit{false})\\
& \rightarrow & \underline{\mathsf{in}(E,\mathsf{if}}(\mathsf{var}(x),\mathsf{if}(\mathsf{var}(y),\mathsf{false},\mathsf{true}),\mathsf{true}))\\
& \rightarrow & \mathsf{if}(\underline{\mathsf{in}(E,\mathsf{var}}(x)),\mathsf{in}(E,\mathsf{if}(\mathsf{var}(y),\mathsf{false},\mathsf{true})),\mathsf{in}(E,\mathsf{true}))\\
& \rightarrow & \underline{\mathsf{if}(\mathit{true}},\mathsf{in}(E,\mathsf{if}(\mathsf{var}(y),\mathsf{false},\mathsf{true})),\mathsf{in}(E,\mathsf{true}))\\
& \rightarrow & \underline{\mathsf{in}(E,\mathsf{if}}(\mathsf{var}(y),\mathsf{false},\mathsf{true}))\\
& \rightarrow & \mathsf{if}(\underline{\mathsf{in}(E,\mathsf{var}}(y)),\mathsf{in}(E,\mathsf{false}),\mathsf{in}(E,\mathsf{true}))\\
& \rightarrow & \underline{\mathsf{if}(\mathit{false}},\mathsf{in}(E,\mathsf{false}),\mathsf{in}(E,\mathsf{true}))\\
& \rightarrow & \underline{\mathsf{in}(E,\mathsf{true})}\\
& \rightarrow & \mathit{true}.
\end{array}
$$

Aux réductions qui concernent l'opérateur in près, cette dérivation est peu éloignée de celle donnée à la section §2.5.3 (sémantique). Considérer $\boldsymbol{D}''$ comme une $(F'' \cup D)$- plutôt qu'une F''-interprétation nous enjoint à omettre la dernière réduction $\mathit{true} \rightarrow \lambda E.\mathit{true} \in D''$.

$$
\begin{array}{lcl}
\varrho(\mathit{true}) & = & \dagger\\
\varrho(\mathit{false}) & = & \dagger\\
\varrho(\mathsf{true}) & = & \mathit{true}\\
\varrho(\mathsf{false}) & = & \mathit{false}\\
\varrho(\mathsf{if}(t_1,t_2,t_3)) & = & \begin{cases} t_2 & \text{si } t_1 = \mathit{true}\\ t_3 & \text{si } t_1 = \mathit{false}\\ \mathsf{if}(\varrho(t_1),t_2,t_3) & \text{sinon} \end{cases}\\
\varrho(\mathsf{call}_n(\varphi_{\mathit{fun}},t_1,\ldots,t_n)) & = & \begin{cases} \mathsf{call}_n(\varphi_{\mathit{fun}},\varrho(t_1),\ldots,t_n) \quad \text{si } \varrho(t_1) \neq \dagger, \text{ sinon}\\ \ldots\\ \mathsf{call}_n(\varphi_{\mathit{fun}},t_1,\ldots,\varrho(t_n)) \quad \text{si } \varrho(t_n) \neq \dagger, \text{ sinon}\\ \mathsf{in}(\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\},t)\\ \qquad \text{pour } \langle \varphi_{\mathit{fun}} = \mathsf{fun}_n(x_1,\ldots,x_n,t)\rangle \in \Sigma'' \end{cases}\\
\varrho(\mathsf{in}(E,\mathsf{true})) & = & \varrho(\mathsf{true})\\
\varrho(\mathsf{in}(E,\mathsf{false})) & = & \varrho(\mathsf{false})\\
\varrho(\mathsf{in}(E,\mathsf{var}(x))) & = & E(x)\\
\varrho(\mathsf{in}(E,\mathsf{if}(t_1,t_2,t_3))) & = & \varrho(\mathsf{if}(\mathsf{in}(E,t_1),\mathsf{in}(E,t_2),\mathsf{in}(E,t_3)))\\
\varrho(\mathsf{in}(E,\mathsf{call}_n(f,t_1,\ldots,t_n))) & = & \varrho(\mathsf{call}_n(f,\mathsf{in}(E,t_1),\ldots,\mathsf{in}(E,t_n)))
\end{array}
$$

Figure 2.14 : Stratégie de réduction dans BOOL avec des schémas réguliers

Trace d'exécution. Cette dérivation a pour trace $\langle\!\langle r_{\mathsf{nand}}, r_{\mathsf{call}_2}, r_{\mathsf{in\,if}}, r_{\mathsf{var}}, r_{\mathsf{if}\,\mathit{true}}, r_{\mathsf{in\,if}}, r_{\mathsf{var}}, r_{\mathsf{if}\,\mathit{false}}, r_{\mathsf{true}}\rangle\!\rangle$. Notons toutefois que la règle $r_{\mathsf{in\,if}}$ ne correspond à rien dans les modèles d'exécution précédents. Dans une implémentation comme celle de BOOL (cf. §2.2), on ne peut pas lui associer un comportement dynamique. Elle exprime simplement la visibilité d'une expression sur un environnement. La règle $r_{\mathsf{in\ call}}$ admet la même remarque.

On peut comparer la trace de ces règles aux actions internes observables τ, dans les systèmes de transitions. De même, la trace r_φ est superflue car elle est toujours immédiatement suivie de r_{call}. La trace *abstraite* de l'exécution précédente est alors simplement $\langle\!\langle r_{\mathsf{call}_2}, r_{\mathsf{var}}, r_{\mathsf{if}\,true}, r_{\mathsf{var}}, r_{\mathsf{if}\,false}, r_{\mathsf{true}} \rangle\!\rangle$.

Parce que l'environnement est très simple dans ce cas particulier, on obtient une trace sensiblement identique à celle d'un schéma algébrique qui incorpore un opérateur pour les variables (cf. §2.5.6, trace d'exécution des variables). Pour modéliser l'accès à une variable qui dépend de l'*histoire* de l'environnement, environnement dont il faut donner les caractéristiques opérationnelles, on fait figurer $r_{\mathsf{var}}(E, x)$ dans la trace abstraite, représentant plus fidèle de la règle $\langle \mathsf{in}(E, \mathsf{var}(x)) \to E(x) \rangle$.

Conclusion du chapitre.

Nous avons proposé une définition abstraite de *modèle d'exécution* (§2.1.2) et nous avons montré sur un exemple comment elle permettait de représenter le comportement d'un programme compilé (§2.2).

Nous avons ensuite indiqué quels étaient les éléments à peser pour trouver un compromis entre la fidélité d'un modèle et son degré d'abstraction (§2.3). En particulier, pour permettre une spécification de plus haut niveau, nous avons proposé deux classes de modèles d'exécution, basées sur des formalismes sémantiques : sémantique naturelle (§2.4) et schémas de programme (§2.5).

En ce qui concerne les schémas de programme, nous avons étendu le type de règles employé dans la sémantique opérationnelle (déf. 2.5, prop. 2.1) afin permettre des modèles plus fidèles (cf. §2.5.9). Nous avons également étendu la définition de stratégie de réduction pour des interprétations discrètes donnée par Courcelle [Cou90a, §3.5] à des interprétations continues (§2.5.4). Nous avons montré les avantages d'une spécification en termes de schémas réguliers plutôt qu'algébriques (§2.5.7) et fourni des procédés de traduction de l'un dans l'autre (§2.5.8, 2.5.9), différents de celui donné par Courcelle dans [Cou86, §9]. L'un en particulier (déf. 2.10) permet des modèles d'exécution plus fidèles. Par ailleurs, la proposition 2.3 simplifie la spécification d'une interprétation.

Une fois posées ces bases concernant l'exécution, nous pouvons décrire au chapitre suivant comment affecter une performance à un programme.

Chapitre 3

Mesure de Performance

Pour formuler une notion d'*optimisation*, il faut au préalable pouvoir *mesurer la performance* des programmes ; c'est l'objet de ce chapitre.

On apprécie la performance d'un programme, son *coût*, en fonction des caractéristiques de son exécution. Néanmoins, nous cherchons plus ici à savoir comparer deux programmes entre eux qu'à attribuer une *quantité* explicite à un programme donné. D'autre part, puisqu'un programme représente non pas *une* mais *un ensemble* d'exécutions possibles, il existe aussi plusieurs moyens de comparer les programmes, en fonction des coûts d'exécution individuels.

Sur le plan pratique, un modèle de coût doit pouvoir prendre en compte aussi bien le temps d'exécution que l'espace mémoire consommé et doit pouvoir être comparé à d'autres expressions du coût.

Organisation du chapitre.

§3.1 La performance d'un programme est chiffrée et comparée à l'aide d'une notion de *coût*. Nous équipons ce coût de propriétés abstraites générales et montrons comment en fabriquer de nouveaux par des opérations de combinaison.

§3.2 Une mesure de *coût dynamique* interprète la trace d'un modèle d'exécution afin de chiffrer le prix de l'évaluation $L\ p\ d$ d'un programme p sur une donnée individuelle d.

§3.3 Ce coût dynamique sert à la construction d'un *coût statique*, qui mesure a priori (en l'absence de données d explicites) l'efficacité *intrinsèque* d'un programme p.

§3.4 Nous donnons quelques exemples de constructions de coûts statiques d'usage courant, ainsi que les moyens d'en fabriquer de nouveaux par combinaison.

§3.5 Nous indiquons quelles *qualités* (propriétés) d'un coût statique sont souhaitables *a priori*, et comment ces qualités sont reliées entre elles.

§3.6 Nous examinons en particulier quelles qualités sont satisfaites par les coûts statiques définis à la section §3.4, et comment ils sont corrélés.

Muni ainsi d'un *modèle de performance*, nous formalisons ensuite au chapitre suivant la notion d'*évaluation partielle*.

3.1 Modèle de coût.

Un *coût d'exécution* mesure quantitativement les *ressources* consommées par un programme lors de son exécution. Les ressources *concrètes* sont le *temps* et l'*espace mémoire*. Plus généralement, un *coût* comptabilise toute *quantité abstraite* mesurée sur un programme comme le nombre d'appels de certaines fonctions ou primitives, le nombre d'accès mémoire, la taille *statique* du programme (par opposition à l'espace mémoire consommé *dynamiquement*) afin de contrôler les optimisations qui font croître la taille du code généré, etc.

3.1.1 Domaine et mesure de coût.

Nous définissons des notions abstraites de domaines, de mesure et de modèles de coût. Ainsi, nous pouvons ensuite couvrir une large gamme de modèles de performance et étudier leur propriétés dans un cadre général.

Définition 3.1. (Domaine de coût)
Un *domaine de coût* $(\mathcal{C}, \preccurlyeq)$ est constitué de :

- un *ensemble de coûts* $\mathcal{C}$
- une *comparaison de coût* $\preccurlyeq$, relation transitive sur $\mathcal{C}$

Nous définissons également la *comparaison stricte* $\prec$, l'*équivalence de coût* $\approx$, et la *non-comparaison* $\lessgtr$:

- $\forall c, c' \in \mathcal{C} \quad c \prec c' \quad \text{ssi} \quad c \preccurlyeq c' \quad \text{et} \quad c \not\succcurlyeq c'$
- $\forall c, c' \in \mathcal{C} \quad c \approx c' \quad \text{ssi} \quad c \preccurlyeq c' \quad \text{et} \quad c \succcurlyeq c'$
- $\forall c, c' \in \mathcal{C} \quad c \lessgtr c' \quad \text{ssi} \quad c \not\preccurlyeq c' \quad \text{et} \quad c \not\succcurlyeq c'$

Un domaine de coût $(\mathcal{C}, \preccurlyeq)$ est qualifié du nom (total, antisymétrique, etc.) des propriétés de $\preccurlyeq$.

Une optimisation va consister en une suite de transformations successives, chacune faisant décroître le coût au sens de $\preccurlyeq$. La transitivité est la propriété fondamentale qui assure que le programme final est effectivement meilleur que le programme initial.

En pratique, la relation $\preccurlyeq$ est généralement un préordre ou un ordre, mais n'est pas nécessairement totale. Notons que si $\preccurlyeq$ peut être ou ne pas être réflexive, en revanche $\prec$ est toujours irréflexive (acyclique). Cela ne signifie pas que $\prec$ correspond à la relation $\preccurlyeq$ privée de l'égalité. Néanmoins, $c \prec c'$ implique $c \preccurlyeq c'$ et $c \neq c'$.

Une *mesure de performance* indique quel coût attribuer à une exécution ou à tout un programme. Nous définissons plus généralement une *mesure de coût* μ sur un ensemble quelconque Q comme une *interprétation* de Q dans $\mathcal{C}$.

Définition 3.2. (Mesure de coût, modèle de coût)
Soient Q un ensemble et $(\mathcal{C}, \preccurlyeq)$ un domaine de coût.

- Une *mesure de coût sur* Q est une application $\mu : Q \to \mathcal{C}$
- Un *modèle de coût de* Q est une structure $(\mathcal{C}, \preccurlyeq, \mu)$

La mesure de coût μ transporte la relation de coût $\preccurlyeq$ sur Q :

- $\forall q, q' \in Q \quad q \preccurlyeq_\mu q' \quad \text{ssi} \quad \mu(q) \preccurlyeq \mu(q')$

On définit de même les relations $\prec_\mu$, $\approx_\mu$ et $\lessgtr_\mu$ sur Q.

- Deux modèles $(\mathcal{C}, \preccurlyeq, \mu)$ et $(\mathcal{C}', \preccurlyeq', \mu')$ de Q sont *équivalents* ssi $\preccurlyeq_\mu = \preccurlyeq'_{\mu'}$

Lorsqu'il n'y a pas ambiguïté sur la mesure, nous omettons les indices μ des relations de coût.

3.1.2 Coûts additifs.

Nous faisons l'hypothèse que le coût total d'une exécution résulte de la composition *additive* de coûts *élémentaires* attribués à chaque opération atomique de la machine d'exécution. Cela vaut pour le temps d'exécution comme pour l'espace consommé (cf. §3.2.2). Pour cela, on équipe le domaine de coût d'une structure algébrique, que l'on relie avec la comparaison de coût.

Définition 3.3. (Domaine de coût algébrique)
Un *domaine de coût algébrique* $(\mathcal{C}, +, ., \preccurlyeq)$ est tel que :

- $(\mathcal{C}, \preccurlyeq)$ est un domaine de coût (c.-à-d. $\preccurlyeq$ est transitive)
- $(\mathcal{C}, +, .)$ est un IR-espace vectoriel

Un coût c a un *signe positif* (resp. *négatif*) ssi $c \succcurlyeq 0$ (resp. $c \preccurlyeq 0$). On note $\mathcal{C}_+ = \{c \in \mathcal{C} \mid c \succcurlyeq 0\}$.

En réalité, une structure de monoïde $(\mathcal{C}, +, 0)$ est suffisante si l'on veut uniquement calculer un coût d'exécution ; l'*associativité* de la loi $+$ est liée à l'associativité de la composition des programmes, et l'élément neutre 0 représente le coût d'exécution trivial, lorsqu'il n'y a aucune opération à effectuer. C'est précisément cette structure qu'emploie Gurr [Gur91] dans sa formalisation en termes de catégories. Comme le remarque également Gurr, la *commutativité*, bien que toujours satisfaite dans les domaines pratiques, est le plus souvent superflue. Seul la définition du *coût moyen* (cf. §3.4.4) n'a pas de sens sans commutativité.

Bien que les machines évoluent (théoriquement) dans un monde discret et que les coûts manipulés soient en général positifs — on peut songer par exemple à $(\mathrm{IN}, +, 0, \leqslant)$ — nous avons choisi de plonger la structure de monoïde dans un IR-espace vectoriel afin de simplifier la présentation. Par ailleurs, cette structure s'impose d'elle-même dans la définition de certains domaines de coût statique (cf. §3.3) ; il faut un IR-espace vectoriel pour pondérer une relation de coût par une *distribution* ou pour *majorer* un coût. Dans la pratique, $\mathcal{C}$ est en fait une puissance de IR, avec sa structure de IR-espace vectoriel, ou bien une puissance de IR^2, si l'on opère sur des encadrements de coûts (afin de pallier à des modèles officieusement approximatifs).

Définition 3.4. (Comparaison additive, homothétique)
Soit $(\mathcal{C}, +, ., \preccurlyeq)$ un domaine de coût algébrique. La relation $\preccurlyeq$ est *additive*[1] ssi elle vérifie l'une des conditions équivalentes suivantes.

- $\forall c_1, c_2, c_3 \in \mathcal{C} \quad c_1 \preccurlyeq c_2 \quad \Rightarrow \quad c_1 + c_3 \preccurlyeq c_2 + c_3$
- $\forall c_1, c_2, c'_1, c'_2 \in \mathcal{C} \quad c_1 \preccurlyeq c_2 \ \wedge \ c'_1 \preccurlyeq c'_2 \quad \Rightarrow \quad c_1 + c'_1 \preccurlyeq c_2 + c'_2$

La relation $\preccurlyeq$ est *homothétique*[2] ssi

- $\forall c_1, c_2 \in \mathcal{C} \quad \forall \alpha \in \mathrm{IR}_+ \backslash \{0\} \quad c_1 \preccurlyeq c_2 \ \Rightarrow \ \alpha.c_1 \preccurlyeq \alpha.c_2$

Pour désigner ces propriétés, nous parlerons d'*additivité*[3], ainsi que de l'élégante *homothéticité*.

La condition d'additivité signifie que la relation de coût est dans une certaine mesure compatible avec la *composabilité* des programmes. Considérons deux programmes p_1 et p_2 équivalents, et de coûts respectifs c_1

[1] Cette propriété est aussi appelée *compatibilité de* $\preccurlyeq$ *avec* $+$.

[2] On qualifie d'habitude de *linéaire* une relation qui, dans notre terminologie, est *additive* et *homothétique*. Nous préférons dissocier les deux concepts car ils interviennent parfois séparemment dans diverses propriétés. En outre, parler de *coût linéaire* prête à confusion. Notons par ailleurs qu'une relation additive est homothétique « sur les entiers » : si $c_1 \preccurlyeq c_2$ alors $n.c_1 \preccurlyeq n.c_2$ pour $n \in \mathrm{IN} \backslash \{0\}$.

[3] Si l'on considérait le domaine de coût simplement un monoïde, l'absence d'inverse nous obligerait à définir la *soustractivité*, propriété réciproque de l'additivité : $\forall c_1, c_2, c_3 \in \mathcal{C} \quad c_1 + c_3 \preccurlyeq c_2 + c_3 \quad \Rightarrow \quad c_1 \preccurlyeq c_2$. Ce qui est vrai pour l'additivité dans les tableaux 3.2, 3.3, 3.4, 3.5 donnés en fin de chapitre l'est encore si l'on ne suppose pas l'existence d'un inverse et l'est aussi pour la soustractivité. Seul le caractère discriminant du coût moyen nécessite additivité et soustractivité.

et c_2 sur une donnée d, avec $c_1 \preccurlyeq c_2$: le programme p_1 est préférable à p_2 sur d. Si p_1 et p_2 sont en réalité des procédures d'un plus gros programme, dont les calculs supplémentaires comptent pour un coût c_3, l'additivité garantit $c_1 + c_3 \preccurlyeq c_2 + c_3$: il est avantageux, au sens de $\preccurlyeq$ et sur la donnée d, de remplacer p_2 par p_1 dans n'importe quel contexte.

3.1.3 Combinaison de coûts.

Une fois que l'on dispose de modèles de coût, on peut les combiner entre eux afin de former de nouveaux modèles, plus complexes. Nous définissons ainsi la *conjonction*, le *produit* et le *produit lexicographique*. Les propriétés des domaines (et en particulier l'indispensable transitivité) sont données à la section §3.6.2.

Conjonction, disjonction.

Définition 3.5. (Coût conjoint)
Le *domaine de coût conjoint* de la famille de domaines $(\mathcal{C}, \preccurlyeq_i)_{i\in I}$ est $(\mathcal{C}, \bigcap_{i\in I} \preccurlyeq_i)$, défini par :

- $\forall c, c' \in \mathcal{C} \quad c \, (\bigcap_{i\in I} \preccurlyeq_i) \, c' \quad \text{ssi} \quad \forall i \in I \quad c \preccurlyeq_i c'$

Le *modèle de coût conjoint* de la famille de modèles $(\mathcal{C}, \preccurlyeq_i, \mu)_{i\in I}$ est $(\mathcal{C}, \bigcap_{i\in I} \preccurlyeq_i, \mu)$.

Soient par exemple deux relations de coûts, l'une $\preccurlyeq_{\mathrm{moy}}$ comparant les programmes selon le coût moyen d'exécution, et l'autre $\preccurlyeq_{\max}$ comparant les programmes selon leur pire cas. Remplacer le programme p par le programme p' lorsque la conjonction $p' \; (\preccurlyeq_{\mathrm{moy}} \cap \preccurlyeq_{\max}) \; p$ est vérifiée assure que p' est meilleur en moyenne que p, sans perte de performance dans le pire cas.

- $p' \; (\preccurlyeq_{\mathrm{moy}} \cap \preccurlyeq_{\max}) \; p \quad \text{ssi} \quad p' \preccurlyeq_{\mathrm{moy}} p \quad \text{et} \quad p' \preccurlyeq_{\max} p.$

Nous définissons dans la suite des relations de coût à l'aide de conjonctions sur des familles infinies. C'est pourquoi nous n'avons pas simplement donné une définition de conjonction binaire.

Définition 3.6. (Disjonction de relations)
La *disjonction* d'une famille $(\preccurlyeq_i)_{i\in I}$ de relations sur $\mathcal{C}$ est $\bigcup_{i\in I} \preccurlyeq_i$ définie par :

- $\forall c, c' \in \mathcal{C} \quad c \, (\bigcup_{i\in I} \preccurlyeq_i) \, c' \quad \text{ssi} \quad \exists i \in I \quad c \preccurlyeq_i c'$

Nous ne parlons pas ici d'un *domaine de coût* $(\mathcal{C}, \bigcup_{i\in I} \preccurlyeq_i)$ car la relation $\bigcup_{i\in I} \preccurlyeq_i$ ainsi définie n'est pas toujours transitive. Néanmoins, la transitivité est parfois préservée lorsque $(\preccurlyeq_i)_{i\in I}$ possède des propriétés d'inclusions (tabl. 3.2).

Coût produit.

Le coût n'est pas nécessairement une quantité scalaire ; il peut comptabiliser séparément plusieurs paramètres, liés de façons diverses au critère de performance. Autrement dit, le coût est à valeur dans un produit d'ensembles. Si chacun d'eux est muni d'une relation de coût, on peut naturellement former la relation produit.

Définition 3.7. (Coût produit)
Le *domaine de coût produit* des domaines $(\mathcal{C}_i, \preccurlyeq_i)_{i\in I}$ est $(\prod_{i\in I} \mathcal{C}_i, \bigotimes_{i\in I} \preccurlyeq_i)$, défini par :

- $\forall (c_i)_{i\in I}, (c'_i)_{i\in I} \in \prod_{i\in I} \mathcal{C}_i \quad (c_i)_{i\in I} \, (\bigotimes_{i\in I} \preccurlyeq_i) \, (c'_i)_{i\in I} \quad \text{ssi} \quad \forall i \in I \quad c_i \preccurlyeq_i c'_i$

La *mesure de coût produit* de la famille de mesures $(\mu_i)_{i\in I}$ sur Q, aussi notée $(\mu_i)_{i\in I}$, est définie par :

- $(\mu_i)_{i\in I} : Q \to \prod_{i\in I} \mathcal{C}_i$ tq $\forall q \in Q \quad (\mu_i)_{i\in I}(q) = (\mu_i(q))_{i\in I}$

La comparaison induite sur Q est alors :

- $(\bigotimes_{i\in I} \preccurlyeq_i)_{(\mu_i)_{i\in I}} = \bigcap_{i\in I} \preccurlyeq_{i,\mu_i}$

Le *modèle de coût produit* des modèles $(\mathcal{C}_i, \preccurlyeq_i, \mu_i)_{i\in I}$ est $(\prod_{i\in I} \mathcal{C}_i, \bigotimes_{i\in I} \preccurlyeq_i, (\mu_i)_{i\in I})$. Si les domaines de coûts $(\mathcal{C}_i, +_i, \cdot_i, \preccurlyeq_i)_{i\in I}$ sont algébriques, l'ensemble de coût produit $\prod_{i\in I} \mathcal{C}_i$ est muni de la structure de $\mathbb{R}$-espace vectoriel produit.

Considérons par exemple deux domaines, l'un $(\mathcal{C}_\mathrm{t}, \preccurlyeq_\mathrm{t})$ chiffrant spécifiquement le *temps* d'exécution, et l'autre $(\mathcal{C}_\mathrm{e}, \preccurlyeq_\mathrm{e})$ l'*espace*, les ressources mémoire consommées. Dans le domaine $(\mathcal{C}_\mathrm{t} \times \mathcal{C}_\mathrm{e}, \preccurlyeq_\mathrm{t} \otimes \preccurlyeq_\mathrm{e})$, un coût d'exécution est meilleur qu'un autre s'il est à la fois meilleur en temps et en espace :

- $(c_\mathrm{t}, c_\mathrm{e}) \ (\preccurlyeq_\mathrm{t} \otimes \preccurlyeq_\mathrm{e}) \ (c'_\mathrm{t}, c'_\mathrm{e})$ ssi $c_\mathrm{t} \preccurlyeq_\mathrm{t} c'_\mathrm{t}$ et $c_\mathrm{e} \preccurlyeq_\mathrm{e} c'_\mathrm{e}$

Le produit est fini en pratique, et s'exprime à l'aide du produit de relations binaires, qui est associatif (à un isomorphisme près) au même titre que le produit cartésien binaire.

Coût produit lexicographique.

Une autre relation produit classique est l'« ordre lexicographique ». Ainsi, le *domaine de coût produit lexicographique* $(\mathcal{C}_\mathrm{t} \times \mathcal{C}_\mathrm{e}, \preccurlyeq_\mathrm{t} \otimes_\mathrm{lex} \preccurlyeq_\mathrm{e})$ privilégie avant tout le temps d'exécution et, à temps égal, compare la consommation mémoire.

Définition 3.8. (Coût produit lexicographique)
Pout tout ensemble $\mathcal{C}$, on note $\preccurlyeq_\mathrm{triv}$ la *comparaison triviale* définie par :

- $\forall c, c' \in \mathcal{C} \quad c \preccurlyeq_\mathrm{triv} c' \quad$ (on a alors $\preccurlyeq_\mathrm{triv} = \approx_\mathrm{triv}$)

Le *domaine de coût produit lexicographique* $(\prod_{i\in I} \mathcal{C}_i, \bigotimes^\mathrm{lex}_{i\in I} \preccurlyeq_i)$ des domaines $(\mathcal{C}_i, \preccurlyeq_i)_{i\in I}$, où I est un intervalle de $\mathbb{N}$, est défini par[4] :

- $\bigotimes^\mathrm{lex}_{i\in I} \preccurlyeq_i = \bigcup_{i\in I}((\bigotimes_{j\in I, j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I, j>i} \approx_{j,\mathrm{triv}})) \cup (\bigotimes_{i\in I} \preccurlyeq_i)$

La comparaison induite par une famille de mesures $(\mu_i)_{i\in I}$ sur un même ensemble Q est alors :

- $(\bigotimes^\mathrm{lex}_{i\in I} \preccurlyeq_i)_{(\mu_i)_{i\in I}} = \bigcup_{i\in I}((\bigcap_{j\in I, j<i} \preccurlyeq_{j,\mu_j}) \cap \prec_i \cap (\bigcap_{j\in I, j>i} \approx_{j,\mathrm{triv}})) \cup (\bigcap_{i\in I} \preccurlyeq_{i,\mu_i})$

Le *modèle de coût produit lexicographique* des modèles $(\mathcal{C}_i, \preccurlyeq_i, \mu_i)_{i\in I}$ est $(\prod_{i\in I} \mathcal{C}_i, \bigotimes^\mathrm{lex}_{i\in I} \preccurlyeq_i, (\mu_i)_{i\in I})$.

On peut interpréter le produit lexicographique en considérant la $i$ème projection sur $\mathcal{C}_i$ comme une approximation du coût « à l'ordre i ». Dans une comparaison, on ne prend en compte l'ordre $i+1$ qu'en cas d'équivalence des coûts d'ordre inférieur ou égal à i.

Coût produit spécifique.

L'ensemble produit $\mathcal{C}_\mathrm{t} \times \mathcal{C}_\mathrm{e}$ peut être muni de comparaisons de coût $\preccurlyeq$ autres que les relations produits $\preccurlyeq_\mathrm{t} \otimes \preccurlyeq_\mathrm{e}$ ou $\preccurlyeq_\mathrm{t} \otimes_\mathrm{lex} \preccurlyeq_\mathrm{e}$. Si $(\mathcal{C}_\mathrm{t}, \preccurlyeq_\mathrm{t}) = (\mathcal{C}_\mathrm{e}, \preccurlyeq_\mathrm{e}) = (\mathbb{R}, \leqslant)$, définissons par exemple $(c_\mathrm{t}, c_\mathrm{e}) \preccurlyeq (c'_\mathrm{t}, c'_\mathrm{e}) \Leftrightarrow \alpha c_\mathrm{t}^2 + c_\mathrm{e} \leqslant \alpha c'^2_\mathrm{t} + c'_\mathrm{e}$; le temps est alors quadratiquement négligeable ou prépondérant devant l'espace, suivant la valeur du réel α. Une autre écriture consiste à reporter sur la mesure μ la combinaison des coûts. On forme pour cela, à partir des modèles $(\mathcal{C}_\mathrm{t}, \preccurlyeq_\mathrm{t}, \mu_\mathrm{t})$ et $(\mathcal{C}_\mathrm{e}, \preccurlyeq_\mathrm{e}, \mu_\mathrm{e})$, un nouveau modèle $(\mathbb{R}, \leqslant, \mu)$ avec $\mu = \alpha\mu_\mathrm{t}^2 + \mu_\mathrm{e}$.

[4] La définition est aussi équivalente à $\bigotimes^\mathrm{lex}_{i\in I} \preccurlyeq_i = \bigcup_{i\in I}((\bigotimes_{j\in I, j<i} \approx_j) \otimes \prec_i \otimes (\bigotimes_{j\in I, j>i} \approx_{j,\mathrm{triv}})) \cup (\bigotimes_{i\in I} \preccurlyeq_i)$.

3.2 Coût dynamique.

Nous examinons dans cette section comment comparer les coûts d'exécution individuels (pour une donnée d fixée) de deux programmes, en fonction de leur trace d'exécution respective. Ce *coût dynamique* sert ensuite de base à la construction de *coûts statiques* (cf. §3.3) qui permettent la comparaison globale de programmes (sur l'ensemble de leur domaine de définition).

3.2.1 Modèle de performance dynamique.

Définition 3.9. (Modèle de performance dynamique)
Soit $M : \mathcal{P} \times \mathcal{D} \to \mathcal{T}$ un modèle d'exécution d'un langage L.

- Un *chiffrage* des traces de $\mathcal{T}$ dans un domaine de coût *dynamique* $(\mathcal{C}, \preccurlyeq)$ est une application $\chi : \mathcal{T} \to \mathcal{C}$
- La *mesure de performance* (ou *de coût*) *dynamique* $\mu : \mathcal{P} \times \mathcal{D} \to \mathcal{C}$ associée au chiffrage χ est $\mu = \chi \circ M$
- Un *modèle de performance* (ou *de coût*) *dynamique* est une structure $(\mathcal{C}, \preccurlyeq, \mu)$

Nous notons $\mu(p\ d)$ l'expression de la mesure de coût μ sur un élément (p, d) de $Dom(\mu) = Dom(L)$.

Il était envisageable d'attribuer un coût « infini » aux programmes qui ne se terminent pas (c'est le choix fait par exmple dans [Ros89]). Nous avons préféré opérer explicitement sur $Dom(L)$, plutôt que sur tout $\mathcal{P} \times \mathcal{D}$, afin d'exclure le cas des programmes indéfinis. Nous échappons ainsi à des problèmes d'incohérence et simplifions également la formulation des coûts statiques.

Il était aussi possible de définir $\mu : \mathcal{P} \times \mathcal{D} \to \mathcal{C}$ directement, sans passer par l'intermédiaire d'un *modèle d'exécution*, en considérant $\mu = M$ et $\mathcal{C} = \mathcal{T}$. Nous avons préféré distinguer les deux concepts d'*exécution* et de *performance* qui, sans être indépendants, ne sont pas non plus confondus. Nous espérons ainsi avoir une vue à la fois intuitive et formelle.

3.2.2 Temps et espace.

Que l'on considère la consommation de ressources *concrètes* ou *abstraites*, il y a principalement deux types élémentaires de coûts dynamiques.

Temps d'exécution. Chaque opération atomique de la machine d'exécution est effectuée en un *temps* qui lui est propre. Le temps d'exécution total est la somme des temps d'exécution de chacune des instructions ; c'est une quantité « additive ». Cela vaut aussi pour le comptage du nombre d'appels d'une fonction.

Le temps d'exécution d'une instruction est une information contextuelle qui dépend de l'état *interne courant* de la machine d'exécution : indicateur, registre, mémoire... Il faut faire figurer cet état dans la trace d'exécution (cf. §2.3.1) afin de chiffrer le temps.

Espace mémoire consommé. L'*espace mémoire* caractérise, à la fin d'une exécution, la taille mémoire *maximum* qu'a nécessité le programme à tous les instants. On peut également le considérer comme une quantité additive en affectant un coût positif aux instructions qui demandent d'accroître la mémoire réservée à l'exécution, et un coût nul aux autres (il n'y a pas de coût négatif quand on libère de la mémoire car on calcule un coût *maximum*).

L'espace mémoire consommé est une information contextuelle qui dépend de l'*histoire* de l'exécution et qui, si elle n'apparaît pas dans les états *internes* de la machine d'exécution, doit néanmoins figurer dans la trace ; il faut alors un état *externe* à la machine pour conserver la taille maximum déjà requise.

Si par exemple l'instruction « `POP` » a toujours un coût spatial nul, il faut en revanche distinguer un « `PUSH` » qui a lieu alors que la pile est au maximum de sa taille depuis le début de l'exécution, et qui demande une taille de pile plus grande, d'un « `PUSH` » qui survient après un désempilage. C'est la même différence en C entre `sbrk` qui accroît effectivement la taille du segment de données réservé au programme par le système d'exploitation, et `malloc` qui réutilise la mémoire déjà allouée mais libérée ou qui appelle lui-même `sbrk` lorsque la mémoire libre est insuffisante.

Parce que l'espace mémoire effectivement consommé est difficile à exprimer, on considère parfois simplement une borne supérieure de l'espace nécessaire, en ne tenant pas compte de la libération des zones allouées ou du glaneur de cellule [Ros89]. Par exemple, dans le cas de LISP, on se contente de compter le nombre de `cons`.

On peut noter que la taille des données n'intervient pas ici car on cherche à exprimer un coût dynamique, c'est-à-dire le coût d'*une* exécution pour *une* donnée fixée. En revanche, elle pourra intervenir (cf. §3.4.9) dans le cadre d'un coût statique (cf. §3.3). Ainsi, une autre quantité également mesurée est la *complexité en taille* [Weg75, LM88], qui chiffre la taille du résultat (en fonction de la taille des données).

Le coût associé à la taille (statique) d'un programme peut être comptabilisé, soit pour chaque exécution (c'est alors un incrément au coût dynamique), soit, puisqu'il est constant, pour un ensemble d'exécution (cf. §4.8.3).

Axiome en guise de conclusion. Dans notre formalisation, la trace seule reste détentrice de toute l'information concernant l'exécution et, par là même, la performance. De sa fidélité dépend la pertinence du modèle.

3.2.3 Performance d'une machine concrète ou abstraite.

Si l'on ne prend pas en compte les contraintes extérieures au processeur, la mesure du temps d'exécution pour une machine concrète est déterminée par le nombre de cycles nécessaires pour exécuter les instructions du programme et le temps de cycle de l'horloge. La mesure de l'espace est le nombre d'octets alloués, la taille consommée par une pile, etc. Pour des machines plus abstraites, les coûts, eux-même abstraits, peuvent être déduits d'une implémentation de cette machine, ou bien peuvent mesurer des quantités de plus haut niveau.

En pratique, un coût $c_{\tilde{\imath}}$ est attribué à la trace élémentaire $\tilde{\imath}$ de chaque instruction i d'une machine, concrète ou abstraite. Le coût total d'une exécution est la somme des coûts individuels de chacune des instructions. Par exemple, nous avons vu que l'exécution du code $\boldsymbol{comp}[\![dec_{\text{nand}}]\!]$ sur une pile qui contient les valeurs successives `1` et `0` (c.-à-d. la compilation de $\texttt{nand}(true, false)$) avait la trace suivante dans M'_{Mach} (cf. §2.3.2) :

$$\langle\!\langle\, \texttt{MOV BP,SP};\ \texttt{MOV AX,(BP}+depl\texttt{)};\ \texttt{CMP AX,0};\ \texttt{JZ}_{\texttt{ZF=0}}\ addr;\ \texttt{MOV AX,(BP}+depl\texttt{)};\ \texttt{CMP AX,0};$$
$$\texttt{JZ}_{\texttt{ZF=1}}\ addr;\ \texttt{MOV AX,1};\ \texttt{JMP}\ addr;\ \texttt{RET 4}\,\rangle\!\rangle$$

Un chiffrage χ_{Mach}, et la mesure μ_{Mach} associée, estiment par exemple le coût de cette trace à :

$$c_{\texttt{MOV}\,reg,reg} + 2\,c_{\texttt{MOV}\,reg,(reg+depl)} + 2\,c_{\texttt{CMP}\,reg,imm} + c_{\texttt{JZ}_{\texttt{ZF}=0}\,addr} + c_{\texttt{JZ}_{\texttt{ZF}=1}\,addr} + c_{\texttt{MOV}\,reg,imm} + c_{\texttt{JMP}\,addr} + c_{\texttt{RET}\,depl}$$

Nous avons noté d'un même symbole un coût constant dans une famille d'instructions.

Dans le cas d'instructions dont le comportement dynamique dépend du contexte, le coût est une fonction des paramètres qui figurent dans la trace. Par exemple, la trace élémentaire « `get_value` $\texttt{X}n(t)$,$\texttt{A}i(t')$ » de l'instruction `get_value`, qui entre autres unifie ses deux arguments, a pour coût d'exécution une quantité $c_{\texttt{get\_value}\ \texttt{X}n,\texttt{A}i}(t, t')$ qui dépend des termes t et t' pointés respectivement par les registres $\texttt{X}n$ et $\texttt{A}i$ (cf. §2.3.1, compromis). En réalité, cela ne permet de mesurer qu'une quantité liée au temps d'exécution. Pour tenir compte également de l'espace mémoire, il faut aussi faire figurer parmi les paramètres la taille maximum déjà atteinte par la pile d'unification [AK90].

3.2.4 Performance d'un langage compilé.

Nous avons vu que si l'on dispose d'une compilation d'un langage L_s dans un langage L_c, alors un modèle d'exécution de L_c détermine un modèle d'exécution de L_s (déf. 2.4). De même, un modèle de performance de L_c détermine un modèle de performance de L_s.

Définition 3.10. (Modèle de performance d'un langage compilé)
Soient $(T_{\mathcal{P}}, T_{\mathcal{D}}, T_{\mathcal{R}})$ une compilation d'un langage $L_s : \mathcal{P}_s \times \mathcal{D}_s \to \mathcal{R}_s$ dans un langage $L_c : \mathcal{P}_c \times \mathcal{D}_c \to \mathcal{R}_c$ et $(\mathcal{C}, \preccurlyeq, \mu_c : \mathcal{P}_c \times \mathcal{D}_c \to \mathcal{C})$ un modèle de performance dynamique de L_c. Le *modèle de performance de L_s compilé* est $(\mathcal{C}, \preccurlyeq, \mu_s)$ défini par $\mu_s = \mu_c \circ (T_{\mathcal{P}}, T_{\mathcal{D}})$.

C'est simplement dire que la performance d'un programme est celle de son code compilé. Ainsi, le coût d'exécution de $\texttt{nand}(true, \mathit{false})$ selon μ_{Comp} est le même que celui indiqué ci-dessus pour μ_{Mach} à la section §3.2.3.

3.2.5 Performance en sémantique naturelle.

Les opérateurs de construction d'arbres de preuve jouent le rôle d'instructions élémentaires de la machine d'exécution en sémantique naturelle. Ils correspondent en pratique à chacune des constructions du langage, permettant ainsi de chiffrer leur performance individuelle.

Interprétation additive.

La justification donnée pour l'additivité à la section §3.1.2 suggère aussi que la mesure de coût est obtenue par composition des coûts des sous-termes. Par exemple, si les coûts des expressions « `b^2` » et « `4*a*c` » sont respectivement c_1 et c_2, le coût global de « `b^2-4*a*c` » est $c_1 + c_2 + c_{\texttt{SUB}}$, où $c_{\texttt{SUB}}$ est le coût spécifique d'une soustraction.

Plus généralement, pour associer un coût à une preuve, on assimile un chiffrage χ à une interprétation « additive » des arbres de preuve dans les coûts : si les arbres sont construits sur une signature F, χ est l'unique homomorphisme de $\boldsymbol{M}(F)$ dans la F-algèbre $\langle \mathcal{C}, (f_{\mathcal{C}})_{f\in F}\rangle$ où les fonctions $f_{\mathcal{C}}$ sont déterminées par un ensemble de coûts $(c_f)_{f\in F}$:

- $\forall f \in F_n \quad \forall c_1, \ldots, c_n \in \mathcal{C} \quad f_{\mathcal{C}}(c_1, \ldots, c_n) = c_f + c_1 + \cdots + c_n$

Par exemple, la trace $M'_{\mathrm{S.Nat}}(\mathtt{nand}(\mathit{true}, \mathit{false})) = \rho_{\mathsf{if}}(\rho_{\mathsf{var}}, \rho_{\mathsf{case\,true}}(\rho_{\mathsf{if}}(\rho_{\mathsf{var}}, \rho_{\mathsf{case\,false}}(\rho_{\mathsf{true}}))))$ (cf. §2.4.4) correspond au coût $\mu_{\mathrm{S.Nat}}(\mathtt{nand}(\mathit{true}, \mathit{false}))$ suivant :

$$2\, c_{\mathsf{if}} + 2\, c_{\mathsf{var}} + c_{\mathsf{case\,true}} + c_{\mathsf{case\,false}} + c_{\mathsf{true}}$$

Si la trace est moins abstraite, si par exemple l'accès à la valeur d'une variable a une trace $\rho_{\mathsf{var}}(E, x)$, le coût élémentaire associé $c_{\mathsf{var}}(E, x)$ dépend aussi de E et x.

L'analyse de coût de Sands [San93] comptabilise le nombre de fois qu'une règle particulière apparaît dans l'arbre de preuve. Dans la formulation précédente, cela revient à décompter dans $(\mathbb{N}, \leqslant)$ un coût élémentaire de 1 pour chaque occurrence de cette règle, et 0 pour les autres règles.

Coût synthétisé.

Cette formulation revient aussi à considérer le coût d'exécution comme un *attribut synthétisé* de l'arbre de preuve. Considérons par exemple la règle $\rho_{\mathsf{if\,true}}$:

$$\frac{E \vdash \mathit{exp}_1 \Rightarrow \mathit{true} \qquad E \vdash \mathit{exp}_2 \Rightarrow v}{E \vdash \mathtt{if}\ \mathit{exp}_1\ \mathtt{then}\ \mathit{exp}_2\ \mathtt{else}\ \mathit{exp}_3 \Rightarrow v}$$

On matérialise le coût c d'exécution d'une expression exp dans un environnement E comme un attribut synthétisé de l'arbre de preuve : un jugement prend alors la forme $E \vdash \mathit{exp} \Rightarrow v : c$.

$$\frac{E \vdash \mathit{exp}_1 \Rightarrow \mathit{true} : c_1 \qquad E \vdash \mathit{exp}_2 \Rightarrow v : c_2}{E \vdash \mathtt{if}\ \mathit{exp}_1\ \mathtt{then}\ \mathit{exp}_2\ \mathtt{else}\ \mathit{exp}_3 \Rightarrow v : c_1 + c_2 + c_{\mathsf{if\,true}}}$$

Le coût est d'une certaine manière « calculé en même temps que le résultat[5] ». Il est plus simple de ne faire figurer que l'*incrément* aux coûts synthétisés ; leur composition est alors implicite.

$$\frac{E \vdash \mathit{exp}_1 \Rightarrow \mathit{true} \qquad E \vdash \mathit{exp}_2 \Rightarrow v}{E \vdash \mathtt{if}\ \mathit{exp}_1\ \mathtt{then}\ \mathit{exp}_2\ \mathtt{else}\ \mathit{exp}_3 \Rightarrow v} : c_{\mathsf{if\,true}}$$

Lorsque le comportement dynamique dépend de la valeur de certains paramètres, une règle est annotée ainsi :

$$\frac{}{E \vdash \mathit{var} \Rightarrow E(\mathit{var})} : c_{\mathsf{var}}(E, \mathit{var})$$

Dans les spécifications qui suivent, une règle sans annotation est supposée de coût nul.

Implémentation formelle.

Voici un exemple de chiffrage des arbres de preuve qui modélise la performance dans l'implémentation donnée à la section §2.2.

[5] C'est un peu comme si un programme était doté de moyens pour simultanément calculer son résultat et estimer son coût d'exécution (déduction faite de cette tâche supplémentaire). C'est ce que réalisent dans une certaine mesure les outils d'*instrumentation* qui analysent dynamiquement le comportement des programmes (profiler) ou, dans un domaine voisin, les simulateurs de circuits intégrés.

- $c_{\mathsf{true}} = c_{\mathsf{false}} = c_{\mathtt{MOV}\,reg,imm}$
- $c_{\mathsf{var}} = c_{\mathtt{MOV}\,reg,(reg+depl)}$
- $c_{\mathsf{if}} = c_{\mathtt{CMP}\,reg,imm}$
- $c_{\mathsf{case\,true}} = c_{\mathtt{JZ}_{\mathtt{ZF=0}}\,addr} + c_{\mathtt{JMP}\,addr}$
- $c_{\mathsf{case\,false}} = c_{\mathtt{JZ}_{\mathtt{ZF=1}}\,addr}$
- $c_{\mathsf{call}\,n} = \begin{cases} \text{si } n = 0 \text{ alors } c_{\mathtt{CALL}\,addr} + c_{\mathtt{RET}} \\ \text{si } n > 0 \text{ alors } (n+1)\,c_{\mathtt{PUSH}\,reg} + c_{\mathtt{CALL}\,addr} + c_{\mathtt{POP}\,reg} + c_{\mathtt{MOV}\,reg,reg} + c_{\mathtt{RET}\,depl} \end{cases}$

Il est possible d'obtenir une même mesure de performance à partir du modèles $M_{\mathrm{S.Nat}}$ (cf. §2.4.4, non-pseudo-déterminisme). Il suffit pour cela que :

- $c_{\mathsf{if}} + c_{\mathsf{case\,true}} = c_{\mathsf{if\,true}}$ et $c_{\mathsf{if}} + c_{\mathsf{case\,false}} = c_{\mathsf{if\,false}}$

L'emploi de règles pseudo-déterministes n'est donc pas indispensable.

Implémentation informelle.

Lorsqu'une spécification en sémantique naturelle détermine à elle seule la machine d'exécution (cf. §2.4.3), on donne un modèle de performance en annotant chacune des règles au moyen d'un coût élémentaire, comme nous l'avons fait ci-dessus. Cela peut être accompagné de la justification informelle qui accrédite la fidélité du modèle d'exécution (cf. §2.3.3).

Si l'on oublie momentanément la description formelle de l'implémentation de BOOL donnée à la section §2.2, on peut par exemple *justifier informellement* les règles annotées de la figure 3.1 ainsi :

- L'évaluation d'une valeur booléenne *immédiate* (`true` ou `false`) a un coût constant c_{bool}.
- L'évaluation d'une variable accède en temps constant c_{var} à une valeur rangée dans l'environnement, implémenté dans une pile comme un tableau d'arguments.
- Le coût de la conditionnelle `if` est $c_{\mathtt{if}}$: il comprend le coût d'un test et, suivant que la condition est vraie ou fausse, un saut avant ou après le traitement de la branche `then` ou `else` correspondante.
- L'appel fonctionnel a un coût c_{fun_n} qui ne dépend que du nombre n d'arguments : il faut empiler la valeur de chaque argument, appeler la fonction, et nettoyer la pile au retour. On peut également préciser que ce coût est linéaire en fonction du nombre d'arguments, excepté pour $n = 0$ afin de matérialiser l'optimisation du cas sans argument : $c_{fun_0} = c_{\mathtt{call}}$ et $c_{fun_n} = c_{\mathtt{call}} + c_{\mathtt{somearg}} + n\,c_{\mathtt{arg}}$ pour $n > 0$.

L'argumentation détermine la pertinence des règles non-pseudo-déterministes, ou le recours éventuel à des règles pseudo-déterministes. Nous donnons un autre exemple de ce style de spécification à la section §4.6.1 (modèle de performance).

Si l'on compare cette mesure de performance à celle de l'implémentation effective, la seule hypothèse supplémentaire concerne le coût d'exécution de l'instruction `JZ` : on a supposé que « $\mathtt{JZ}_{\mathtt{ZF=1}}\ addr$ » a un comportement dynamique similaire au test « $\mathtt{JZ}_{\mathtt{ZF=0}}$ » suivi d'un saut « $\mathtt{JMP}\ addr$ », c'est-à-dire que $c_{\mathtt{JZ}_{\mathtt{ZF=1}}\,addr} = c_{\mathtt{JZ}_{\mathtt{ZF=0}}\,addr} + c_{\mathtt{JMP}\,addr}$. Cette identité n'est en fait qu'« approchée » dans le cas général ; c'est le cas en particulier pour la mesure du temps d'exécution dans le 8086 qui chiffre en nombre de cycles $c_{\mathtt{JZ}_{\mathtt{ZF=1}}} = 16$, $c_{\mathtt{JZ}_{\mathtt{ZF=0}}} = 4$, $c_{\mathtt{JMP}} = 15$. Cependant, c'est la mesure de performance simplificatrice que nous adoptons pour BOOL dans la suite du document.

$$\frac{}{E \vdash \texttt{true} \Rightarrow \mathit{true}} : c_{bool} \qquad \frac{}{E \vdash \texttt{false} \Rightarrow \mathit{false}} : c_{bool} \qquad \frac{}{E \vdash \mathit{var} \Rightarrow E(\mathit{var})} : c_{var}$$

$$\frac{E \vdash \mathit{exp}_1 \Rightarrow \mathit{true} \qquad E \vdash \mathit{exp}_2 \Rightarrow v}{E \vdash \texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \Rightarrow v} : c_{\texttt{if}} \qquad \frac{E \vdash \mathit{exp}_1 \Rightarrow \mathit{false} \qquad E \vdash \mathit{exp}_3 \Rightarrow v}{E \vdash \texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \Rightarrow v} : c_{\texttt{if}}$$

$$\frac{\begin{array}{c} E \vdash \mathit{exp}_1 \Rightarrow v_1 \quad \ldots \quad E \vdash \mathit{exp}_n \Rightarrow v_n \\ E(\mathit{fun}) = (\mathit{var}_1, \ldots, \mathit{var}_n, \mathit{exp}) \qquad E + \{\mathit{var}_1 \mapsto v_1, \ldots, \mathit{var}_n \mapsto v_n\} \vdash \mathit{exp} \Rightarrow v \end{array}}{E \vdash \mathit{fun}(\mathit{exp}_1, \ldots, \mathit{exp}_n) \Rightarrow v} : c_{\mathit{fun}_n}$$

Figure 3.1 : Sémantique avec coûts des expressions de BOOL

Domaine de coût abstrait.

Nous n'avons pas, jusqu'à présent, explicité concrètement de domaine de coût $(\mathcal{C}, +, ., \preccurlyeq)$ pour nos exemples. Ce n'est pas toujours indispensable ; le choix effectif d'un domaine et l'affectation de valeurs explicites aux symboles c_i peuvent être différés si l'on dispose de *propriétés abstraites* sur les coûts élémentaires.

Puisqu'un coût d'exécution est formé par composition additive de coûts élémentaires $(c_i)_{i \in I}$ de $\mathcal{C}$ attribués aux opérations de la machine d'exécution, on peut réciproquement construire un *domaine de coût abstrait* comme l'ensemble des combinaisons linéaires *finies* des $(c_i)_{i \in I}$. On le note $\sum_{i \in (I)} \mathbb{R} c_i$ en insistant à nouveau sur le fait que la somme reste finie.

Quant à la relation de coût $\preccurlyeq$ sur $\mathcal{C}$, elle peut être *partiellement* spécifiée en prolongeant par transitivité, et éventuellement, réflexivité, additivité, etc. un ensemble de relations ponctuelles sur des combinaisons de $(c_i)_{i \in I}$. On peut ainsi obtenir des propriétés générales sur une classe de domaines qui vérifient un ensemble d'axiomes. Cette notion est similaire à celle de classe équationnelle d'interprétation dans les algèbres (cf. §5.1.2).

Par exemple pour BOOL, nous considérons un domaine de coût algébrique abstrait $(\mathcal{C}, +, ., \preccurlyeq)$ que nous supposons additif, et la mesure de performance $\mu : \mathrm{Dec} \times (\mathrm{Fun} \times \mathrm{BoolVal}^*) \to \mathcal{C}$ définie par composition des coûts élémentaires $c_{bool}, c_{var}, c_{\texttt{if}}, c_{fun_n} \in \mathcal{C}$ selon les régles de sémantique naturelle de la figure 3.1. Nous faisons l'hypothèse supplémentaire que les coûts élémentaires vérifient :

$$0 \prec c_{bool} \prec c_{var} \preccurlyeq c_{\texttt{if}} \prec c_{fun_n} \prec c_{fun_m} \quad \text{pour} \quad 0 \preccurlyeq n < m$$

Reprenons le programme `nand`.

```
fun nand(X,Y) = if X then (if Y then false else true) else true
```

Ses coûts dynamiques sont :

$$\begin{aligned}
\mu(\texttt{nand}\,(\mathit{true}, \mathit{true})) &= 2\,c_{\texttt{if}} + 2\,c_{var} + c_{bool} \\
\mu(\texttt{nand}\,(\mathit{true}, \mathit{false})) &= 2\,c_{\texttt{if}} + 2\,c_{var} + c_{bool} \\
\mu(\texttt{nand}\,(\mathit{false}, \mathit{true})) &= c_{\texttt{if}} + c_{var} + c_{bool} \\
\mu(\texttt{nand}\,(\mathit{false}, \mathit{false})) &= c_{\texttt{if}} + c_{var} + c_{bool}
\end{aligned}$$

Dans tout domaine de coût algébrique additif qui vérifie les relations ci-dessus, on peut par exemple comparer :

$$\mu(\mathtt{nand}\,(\mathit{false}, \mathit{true})) \prec \mu(\mathtt{nand}\,(\mathit{true}, \mathit{false}))$$

Nous conservons ces hypothèses abstraites sur BOOL dans toute la suite de ce document.

Domaine de coût concret.

Voici un exemple concret de domaine de coût algébrique additif total sur BOOL :

- $(\mathcal{C}, +, ., \preccurlyeq) = (\mathsf{IR}, +, ., \leqslant)$; l'ordre $\leqslant$ satisfait l'hypothèse d'additivité
- $c_{bool} = 4$
- $c_{var} = 17$
- $c_{\mathtt{if}} = 22$
- $c_{fun_0} = 27$ et $c_{fun_n} = 11\,n + 52$ pour $n > 0$

Nous avons déduit ces quantités du nombre de cycles des instructions élémentaires du 8086 [DG82], avec une approximation pour le cas de $c_{\mathtt{if}}$ (cf. « implémentation informelle » ci-dessus). Elles vérifient bien les hypothèses du domaine de coût abstrait ci-dessus (les comparaisons entre coûts élémentaires).

3.2.6 Performance dans les schémas de programme.

Règles de réécriture.

En ce qui concerne les schémas de programme, le chiffrage à l'aide de coûts de la trace d'une dérivation est similaire au chiffrage de la trace d'une machine concrète ou abstraite. Le coût total d'une exécution est la somme des coûts élémentaires c_r attribués à chacune des règles r.

Par exemple, la trace $\langle\!\langle r_{\mathsf{call}_2}, r_{\mathsf{var}}, r_{\mathsf{if}\,true}, r_{\mathsf{var}}, r_{\mathsf{if}\,false}, r_{\mathsf{true}}\rangle\!\rangle$ de l'exécution $\mathtt{nand}\,(true, \mathit{false})$ (cf. §2.5.9, exemple de conversion) a pour coût d'exécution $c_{\mathsf{call}_2} + 2\,c_{\mathsf{var}} + c_{\mathsf{if}\,true} + c_{\mathsf{if}\,false} + c_{\mathsf{true}}$. On modélise ainsi une mesure de performance similaire à celle obtenue en sémantique naturelle, au coût c_{call_2} de l'appel explicite près.

Interprétation.

Il est également possible de définir un coût d'exécution directement au niveau de l'interprétation, sans passer par l'intermédiaire de règles de réécriture. Pour cela, on prend comme trace (peu naturelle) d'une exécution l'arbre algébrique plus petite solution du schéma (cf. §2.5.2). Le modèle d'exécution est donc défini par :

- $M : \mathcal{P} \times \mathcal{D} \to \boldsymbol{M}(F, D)$ telle que $\forall (p, d) \in \mathcal{P} \times \mathcal{D} \quad M(p\ d) = \varphi(d_1, \ldots, d_n)_{\Sigma, \boldsymbol{M}(F,D)}$
  avec $\Sigma = T_{\mathcal{P}}(p)$ et $(\varphi, (d_1, \ldots, d_n)) = T_{\mathcal{D}}(d)$

De même que nous avons annoté les règles de sémantique naturelle pour qu'elles calculent aussi des coûts (cf. §3.2.5, coût synthétisé), nous désirons équiper les opérateurs $(f_{\boldsymbol{D}})_{f \in F}$ de moyens similaires afin qu'évaluation et mesure de performance soient simultanées. Dans le contexte algébrique, cela se traduit par une interprétation de l'arbre $\varphi(d_1, \ldots, d_n)_{\Sigma, \boldsymbol{M}(F,D)}$, interprétation qui mêle la sémantique $\boldsymbol{D}$ et le chiffrage des coûts.

Définition 3.11. (Interprétation combinée « sémantique et coût »)
Soient $\boldsymbol{D} = \langle (D_s, \leqslant_s, \perp_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{D}})_{f\in F}\rangle$ une F-interprétation et $\mathcal{C}$ un ensemble de coût. L'*algèbre combinée « sémantique et coût »* $\boldsymbol{D}_{\mathcal{C}}$ est définie par une famille de fonctions $(\chi_f)_{f\in F}$ telles que :

- $\boldsymbol{C} = (C, \leqslant_{\mathcal{C}}, \perp_{\mathcal{C}})$ est le domaine primitif construit sur $\mathcal{C}$: $C = \mathcal{C}_{\perp_{\mathcal{C}}} = \mathcal{C} \cup \{\perp_{\mathcal{C}}\}$ et $\leqslant_{\mathcal{C}}$ est un ordre plat.
- $\boldsymbol{D}_{\mathcal{C}} = \langle (D_{s,\mathcal{C}}, \leqslant_{s,\mathcal{C}}, \perp_{s,\mathcal{C}})_{s\in\mathcal{S}}, (f_{\boldsymbol{D}_{\mathcal{C}}})_{f\in F}\rangle$
- $D_{s,\mathcal{C}} = D_s \times C$
- $\leqslant_{s,\mathcal{C}} = \leqslant_s \otimes \leqslant_{\mathcal{C}}$
- $\perp_{s,\mathcal{C}} = (\perp_s, \perp_{\mathcal{C}})$
- $\forall f \in F_{s_1\ldots s_n s} \quad \forall((d_1,c_1),\ldots,(d_n,c_n)) \in D_{s_1,\mathcal{C}} \times \cdots \times D_{s_n,\mathcal{C}}$
  $f_{\boldsymbol{D}_{\mathcal{C}}}((d_1,c_1),\ldots,(d_n,c_n)) = (f_{\boldsymbol{D}}(d_1,\ldots,d_n), \mu_f(d_1,\ldots,d_n;c_1,\ldots,c_n))$
- $\forall f \in F_{s_1\ldots s_n s} \quad \chi_f : D_{s_1} \times \cdots \times D_{s_n} \times C^n \to C$

C'est une interprétation continue si les fonctions $(\chi_f)_{f\in F}$ sont continues.

Intuitivement, le chiffrage $\chi_f(d_1,\ldots,d_n;c_1,\ldots,c_n)$ représente le coût nécessaire à l'évaluation du terme $f(t_1,\ldots,t_n)$ sachant que l'évaluation du terme t_i retournerait la valeur d_i, avec un coût d'évaluation c_i. Considérons par exemple le cas de $\mathsf{if}(t_1,t_2,t_3)$ dont la sémantique consiste à évaluer t_1 puis, selon le résultat, à évaluer t_2 ou t_3. En nous appuyant sur l'implémentation de BOOL, nous pouvons définir :

- $\chi_{\mathsf{if}}(v_1,v_2,v_3\,;c_1,c_2,c_3) = \begin{cases} c_1 + c_{\texttt{CMP}\,reg,imm} + c_{\texttt{JZ}_{\texttt{ZF}=0}\,addr} + c_2 + c_{\texttt{JMP}\,addr} & \text{si } v_1 = true \\ c_1 + c_{\texttt{CMP}\,reg,imm} + c_{\texttt{JZ}_{\texttt{ZF}=1}\,addr} + c_3 & \text{si } v_1 = false \\ \perp_{\mathcal{C}} & \text{si } v_1 = \perp \end{cases}$

Dans tous les cas, il faut évaluer la condition t_1, opération de coût c_1 ; le choix d'un coût c_2 ou c_3 dépend de la valeur v_1 retournée par l'évaluation de t_1 ; s'ajoutent à cela des coûts constants qui dépendent ou non de v_1.

En fait, l'« exécution de l'opérateur f » dans un terme $f(t_1,\ldots,t_n)$ a deux « activités ». D'une part elle évalue un nombre de fois $N_{f,i}$ chacun de ses sous-termes t_i, nombre dépendant des résultats intermédiaires $d_{i_1},\ldots,d_{i_m}$ déjà calculés, mais fini et en pratique égal à 0 ou 1 ; ces évaluations comptent chacune pour un coût c_i. D'autre part, elle effectue ses propres opérations sur les données renvoyées par l'évaluation des sous-termes, opérations qui participent pour un coût total c_f dépendant aussi des résultats intermédiaires. C'est pourquoi le chiffrage χ_f adopte en pratique la forme suivante.

Définition 3.12. (Interprétation combinée « sémantique et coût algébrique »)
Soit $\boldsymbol{D} = \langle (D_s, \leqslant_s, \perp_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{D}})_{f\in F}\rangle$ une F-interprétation et $(\mathcal{C}, +, ., \preccurlyeq)$ est un domaine de coût algébrique. L'*algèbre combinée « sémantique et coût algébrique »* $\boldsymbol{D}_{\mathcal{C}}$ est définie par deux familles de fonctions $(c_f)_{f\in F}$ et $(N_{f,i})_{n\in\mathbb{N}, f\in F_n, i\in[n]}$ telles que :

- $(\mathcal{C}, +, ., \preccurlyeq)$ est étendu en un domaine tel que $\perp_{\mathcal{C}}$ est absorbant pour les lois « $+$ » et « . », et minimum pour la relation « $\preccurlyeq$ ».
- $\boldsymbol{D}_{\mathcal{C}}$ est l'interprétation construite par la définition 3.11 avec $\forall f \in F_{s_1\ldots s_n s} \quad \forall i \in [n]$:
- $c_f : D_{s_1} \times \cdots \times D_{s_n} \to C$
- $N_{f,i} : D_{s_1} \times \cdots \times D_{s_n} \to \mathbb{N}_{\perp}$ muni d'un ordre plat
- $\forall (d_1,\ldots,d_n) \in D_{s_1} \times \cdots \times D_{s_n} \quad \forall c_1,\ldots,c_n \in C^n$
  $\chi_f(d_1,\ldots,d_n;c_1,\ldots,c_n) = c_f(d_1,\ldots,d_n) + N_{f,1}(d_1,\ldots,d_n)\,c_1 + \cdots + N_{f,n}(d_1,\ldots,d_n)\,c_n$

C'est un interprétation continue si les fonctions $(c_f)_{f\in F}$ et $(N_{f,i})_{n\in\mathbb{N}, f\in F_n, i\in[n]}$ sont continues.

Dans le cas l'opérateur if, cela se traduit par :

- $c_{\mathsf{if}}(v_1, v_2, v_3) = c_{\texttt{CMP}\,reg,imm} + \begin{cases} c_{\texttt{JZ}_{\texttt{ZF}=0}\,addr} + c_{\texttt{JMP}\,addr} & \text{si } v_1 = true \\ c_{\texttt{JZ}_{\texttt{ZF}=1}\,addr} & \text{si } v_1 = false \\ \perp_{\mathcal{C}} & \text{si } v_1 = \perp \end{cases}$
- $N_{\mathsf{if},1}(v_1, v_2, v_3) = 1$
- $N_{\mathsf{if},2}(v_1, v_2, v_3) =$ si $v_1 = true$ alors 1 sinon 0
- $N_{\mathsf{if},3}(v_1, v_2, v_3) =$ si $v_1 = false$ alors 1 sinon 0

Dans le cas d'un opérateur $f \in F_n$ interprété comme une fonction f_D stricte, on a $\forall i \in [n] \quad N_{f,i} = 1$.

Si ce type de formulation convient bien pour estimer le coût d'une expression ne comportant pas d'inconnues, il pose en revanche un problème de fidélité pour une expression $t \in M(F \cup \Phi, D)$ car les appels procéduraux $\varphi(t_1, \ldots, t_n)$ n'apparaissent plus dans l'arbre infini $t_{\Sigma, M(F,D)}$. De même que nous avions rajouté à la signature un opérateur var afin de modéliser l'accès à une variable, nous ajoutons cette fois des opérateurs call_n. La traduction $T_{\mathcal{P}}$ des programmes en un schéma traduit alors un appel fonctionnel non pas en $\varphi(t_1, \ldots, t_n)$, mais en $\mathsf{call}_n(\varphi(t_1, \ldots, t_n))$. Un arbre $t_{\Sigma, M(F,D)}$ conserve ainsi la trace des appels. Parce que le dépliage qui intervient dans la construction de $t_{\Sigma, M(F,D)}$ *copie* les paramètres des inconnues, il faut tout de même affiner cette formulation avec une notion de partage (par exemple avec des moyens évoqués dans [BL79]) lorsque l'on veut modéliser une implémentation où l'appel fonctionnel n'évalue qu'une fois au plus ses arguments.

C'est aussi pour cela que cette présentation convient davantage pour une spécification à l'aide de schémas réguliers (cf. §2.5.7, 2.5.8, 2.5.9) car ils explicitent dans la syntaxe à la fois l'appel procédural et le partage des arguments. Dans le cas de l'appel par valeur, on a par exemple $\forall i \in [n+1] \quad N_{\mathsf{call}_n, i} = 1$.

On peut rapprocher cette formulation de celle qu'emploie Rosendahl pour l'analyse de complexité automatique [Ros89]. Il transforme aussi un programme original pour en fabriquer une version qui compte le nombre de pas d'exécution (tous les coûts c_f sont égaux à 1).

3.3 Coût statique.

La performance d'un programme dépend de la valeur de ses paramètres. S'ils sont encore inconnus, on peut parfois donner des bornes au coût d'exécution. Dans le cas où l'on dispose d'un modèle de distribution de ces paramètres, on peut également fournir une estimation statistique.

L'*analyse d'algorithme* est une discipline qui s'intéresse à la mesure de tels coûts. Elle rend compte avec exactitude de la complexité des algorithmes. Bien qu'elle puisse dans certains cas être automatisée [LM88, Ros89, FSZ89, FSZ91], elle ne sait pas conclure de manière systématique. Elle est de toute façon limitée par l'indécidabilité de la terminaison des programmes, car un résultat de complexité en détient nécessairement la preuve.

L'objectif de l'*optimisation* est de trouver un (ou le) meilleur programme possible. Nous limitons donc notre ambition à *comparer qualitativement* des programmes plutôt qu'à *expliciter quantitativement* la nature de leur mérites respectifs.

3.3.1 Modèle de coût statique.

Comparer des programmes de domaines différents n'a pas beaucoup de sens. Un programme n'est pas meilleur parce qu'il est défini là où un autre ne l'est pas : ils représentent chacuns deux fonctions différentes. Même si l'on restreint la comparaison à l'intersection des domaines de définition, cela ne suffit pas à lever certaines incohérences. La transitivité en particulier n'y résiste pas. C'est pourquoi le modèle de coût statique compare les programmes sur un même ensemble de données $D \subset \mathcal{D}$ fixé, un *support*, inclus dans leur domaine de définition[6].

Définition 3.13. (Support)
Soit $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ un langage de programmation.

- Un *support* D est un sous-ensemble de $\mathcal{D}$.
- $\mathcal{P}_D = \{p \in \mathcal{P} \mid D \subset Dom(p)\}$ est l'ensemble des programmes de *support* D.

Pour comparer un programme p à un programme q sur un support D, on compare *dans leur ensemble* les coûts $\mu(p\ d)_{d\in D}$ et $\mu(q\ d)_{d\in D}$. Autrement dit, une comparaison de coût statique met en relation des applications $\mu(p)$ et $\mu(q)$ de $\mathcal{C}^D$ qui représentent l'ensemble des mesures de coût individuelles de p et q.

Définition 3.14. (coût statique)
Soit $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ un langage, $D \subset \mathcal{D}$ un support et $(\mathcal{C}, \preccurlyeq, \mu)$ un modèle de performance dynamique de L.

- La *mesure de performance statique* est $\hat{\mu} : \mathcal{P} \to (\mathcal{D} \to \mathcal{C})$ tq $\forall p \in \mathcal{P} \quad \forall d \in Dom(p) \quad \hat{\mu}(p)(d) = \mu(p\ d)$

On identifie dans l'écriture μ et sa curryfication $\hat{\mu}$.

- Un *domaine de coût statique de support D* est un domaine de coût $(\mathcal{C}^D, \preccurlyeq_D)$
- Une *mesure de coût statique de support D* est $\mu_D : \mathcal{P}_D \to \mathcal{C}^D$ définie par $\forall p \in \mathcal{P}_D \quad \mu_D(p) = \hat{\mu}(p)_{|D}$
- Un *modèle de coût statique de support D* est une structure $(\mathcal{C}^D, \preccurlyeq_D, \mu_D)$

On définit aussi :

- Un *domaine général de coût statique* est une famille de domaines $(\mathcal{C}^D, \preccurlyeq_D)_{D\subset\mathcal{D}}$
- Un *modèle général de coût statique* est une famille de modèles $(\mathcal{C}^D, \preccurlyeq_D, \mu_D)_{D\subset\mathcal{D}}$
- Un *coût statique général de supports finis* est une famille $(\preccurlyeq_D)_{D\,\text{fini}\subset\mathcal{D}}$

On employera aussi le terme de *modèle de performance* pour désigner un modèle général de coût statique. Lorsque qu'il n'y a pas ambiguïté sur le support, nous omettons l'indice D des relations de coût. Un domaine ou modèle de coût général $(\mathcal{C}^D, \preccurlyeq_D)_{D\subset\mathcal{D}}$ est qualifié du nom (additif, antisymétrique, etc.) des propriétés de *chacune* des relations $(\preccurlyeq_D)_{D\subset\mathcal{D}}$.

L'existence d'un support n'est pas une contrainte importune puisqu'en pratique ce sont les programmes équivalents, et donc de même domaine, qui nous intéressent. Éventuellement, une transformation paresseuse $T(p) = p'$ peut élargir strictement un domaine de définition : $Dom(p) \subsetneq Dom(p')$, mais la comparaison entre p et p' doit impérativement se borner au support initial $Dom(p)$. Si ce n'était pas le cas, le domaine ajouté $(Dom(p') \setminus Dom(p))$ pourrait être constitué de données de faibles coûts d'exécution, réduisant strictement un coût moyen global ($p' \prec p$) mais augmentant celui de la restriction de p' au domaine de p ($p \prec p'_{|Dom(p)}$) ; ce serait l'effet *contraire* d'une optimisation.

Deux programmes, même équivalents, ne sont pas nécessairement comparables car ils peuvent avoir des performances relatives variables selon les données qu'on leur fournit. C'est pourquoi la relation de coût sur les programmes est généralement partielle.

[6] Le contexte lève généralement l'ambiguïté de notation qui existe entre un support $D \subset \mathcal{D}$, et un domaine ou ensemble de domaines $D = (D_s)_{s\in\mathcal{S}}$.

3.3.2 Procédés de construction de coûts statiques.

Il y a plusieurs moyens de construire une relation de coût statique $\preccurlyeq_D$ sur un support D donné :

- déduire $\preccurlyeq_D$ d'un coût dynamique $\preccurlyeq$,
- déduire $\preccurlyeq_D$ d'un autre coût statique $\preccurlyeq'_D$,
- déduire $\preccurlyeq_D$ d'une famille de coûts $(\preccurlyeq'_{D'})_{D' \subset \mathcal{D}}$,
- combiner des coûts statiques connus.

Tout coût statique ainsi construit provient en définitive d'un ou de plusieurs coûts dynamiques. Nous formalisons ici ces *procédés* de construction. La section suivante (§3.4) en donne des exemples pratiques.

Coût statique déduit d'un coût dynamique.

Soit $(\mathcal{C}, \preccurlyeq, \mu)$ un modèle de coût dynamique et D un support. Nous construisons des modèles de coût statique $(\mathcal{C}^D, \preccurlyeq_D, \mu_D)$ de support D. Si $(\mathcal{C}, +, .)$ est un espace vectoriel, on considère l'espace vectoriel $(\mathcal{C}^D, +, .)$ des applications de D dans $(\mathcal{C}, +, .)$. Comme il est d'usage, nous notons de manière identique les lois sur $\mathcal{C}^D$ et sur $\mathcal{C}$, et 0 la fonction nulle de $\mathcal{C}^D$. Notons que si un coût dynamique $\preccurlyeq$ est additif, il n'est pas garanti que $\preccurlyeq_D$ le soit aussi.

Définition 3.15. (Construction de coût statique à partir d'un coût dynamique)
Soient $\mathcal{C}$ un ensemble de coûts et D un support. Un *procédé de construction de coût statique à partir d'un coût dynamique* est une application $\texttt{d-stat}_D$ telle que :

- $\texttt{d-stat}_D : \mathfrak{P}(\mathcal{C} \times \mathcal{C}) \to \mathfrak{P}(\mathcal{C}^D \times \mathcal{C}^D)$

On note $\preccurlyeq_{\texttt{d-stat},D}$ la relation de coût $\texttt{d-stat}_D(\preccurlyeq)$ sur $\mathcal{C}^D$, déduite d'un domaine de coût $(\mathcal{C}, \preccurlyeq)$ et d'un support D.

La notation $\not\preccurlyeq_{\texttt{stat},D}$ est ambiguë ; elle peut désigner $\texttt{stat}_D(\not\preccurlyeq)$ ou $\neg(\texttt{stat}_D(\preccurlyeq))$, qui n'ont a priori aucun rapport. Nous convenons de noter :

- $\not\preccurlyeq_{\texttt{stat},D}$, négation de $\preccurlyeq_{\texttt{stat},D}$, par opposition à $(\not\preccurlyeq)_{\texttt{stat},D}$
- $\succcurlyeq_{\texttt{stat},D}$, réciproque de $\preccurlyeq_{\texttt{stat},D}$, par opposition à $(\succcurlyeq)_{\texttt{stat},D}$
- $\prec_{\texttt{stat},D} = \preccurlyeq_{\texttt{stat},D} \cap \not\succcurlyeq_{\texttt{stat},D}$, par opposition à $(\prec)_{\texttt{stat},D}$
- $\approx_{\texttt{stat},D} = \preccurlyeq_{\texttt{stat},D} \cap \succcurlyeq_{\texttt{stat},D}$, par opposition à $(\approx)_{\texttt{stat},D}$.

Cette convention s'applique à $\texttt{d-stat}$, ainsi qu'aux procédés $\texttt{s-stat}$ et $\texttt{g-stat}$ qui suivent.

Coût statique déduit d'un coût statique.

Définition 3.16. (Construction de coût statique à partir d'un coût statique)
Soient $\mathcal{C}$ un ensemble de coûts et D un support. Un *procédé de construction de coût statique à partir d'un coût statique* est une application $\texttt{s-stat}_D$ telle que :

- $\texttt{s-stat}_D : \mathfrak{P}(\mathcal{C}^D \times \mathcal{C}^D) \to \mathfrak{P}(\mathcal{C}^D \times \mathcal{C}^D)$

On note $\preccurlyeq_{\texttt{s-stat},D}$ la relation de coût $\texttt{s-stat}_D(\preccurlyeq_D)$ sur $\mathcal{C}^D$ de support D, déduite d'un coût statique $\preccurlyeq_D$ sur $\mathcal{C}^D$ de support D.

Coût statique déduit d'un coût statique général.

Définition 3.17. (Construction de coût statique à partir d'un coût statique général)
Soient $\mathcal{C}$ un ensemble de coûts et $\mathcal{D}$ un ensemble de données. Un *procédé de construction de coût statique à partir d'un coût statique général* est une application g-stat telle que :

- g-stat : $\bigcup_{D\subset\mathcal{D}} \mathfrak{P}(\mathcal{C}^D \times \mathcal{C}^D)^D \to \bigcup_{D\subset\mathcal{D}} \mathfrak{P}(\mathcal{C}^D \times \mathcal{C}^D)^D$

Un *procédé de construction de coût statique à partir d'un coût statique général de supports finis* est :

- g-stat : $\bigcup_{D \text{ fini}\subset\mathcal{D}} \mathfrak{P}(\mathcal{C}^D \times \mathcal{C}^D)^D \to \bigcup_{D\subset\mathcal{D}} \mathfrak{P}(\mathcal{C}^D \times \mathcal{C}^D)^D$

On note $\preccurlyeq_{\text{g-stat},D}$ la relation de coût $\texttt{g-stat}_D((\preccurlyeq_{D'})_{D'\subset\mathcal{D}})$ sur $\mathcal{C}^D$ de support D, déduite d'une famille de coût statique $(\preccurlyeq_{D'})_{D'\subset\mathcal{D}}$.

Combinaison de coûts statiques.

Nous avons indiqué à la section §3.1.3 comment construire de nouveaux domaines de coûts par intersection ou par produit. Pour le coût statique, il faut étendre ces définitions au cas d'une famille de domaines.

Définition 3.18. (Coût statique combiné)
Soit $((\mathcal{C}^D, \preccurlyeq_{i,D})_{D\subset\mathcal{D}})_{i\in I}$ une famille de domaines généraux de coût statique.

- Le domaine général de *coût statique conjoint* est $(\mathcal{C}^D, \bigcap_{i\in I} \preccurlyeq_{i,D})_{D\subset\mathcal{D}}$.

On note également :

- $\bigcup_{i\in I}((\preccurlyeq_{i,D})_{D\subset\mathcal{D}}) = (\bigcup_{i\in I} \preccurlyeq_{i,D})_{D\subset\mathcal{D}}$

Soit $((\mathcal{C}_i^D, \preccurlyeq_{i,D})_{D\subset\mathcal{D}})_{i\in I}$ une famille de domaines généraux de coût statique.

- Le domaine général de *coût statique produit* est $((\prod_{i\in I} \mathcal{C}_i)^D, \bigotimes_{i\in I} \preccurlyeq_{i,D})_{D\subset\mathcal{D}}$.
- Le domaine général de *coût statique produit lexicographique* est $((\prod_{i\in I} \mathcal{C}_i)^D, \bigotimes_{i\in I}^{\text{lex}} \preccurlyeq_{i,D})_{D\subset\mathcal{D}}$.

Nous confondons les ensembles isomorphes $(\prod_{i\in I} \mathcal{C}_i)^D \simeq \prod_{i\in I} \mathcal{C}_i^D$.

3.4 Constructions de coûts statiques.

Cette section regroupe quelques exemples de construction de coûts statiques. Ce sont essentiellement les notions ordinaires de « meilleur *partout*, *presque partout*, *en moyenne*, *dans le pire cas* », ainsi que les *majorations* et comparaisons *asymptotiques*. Leurs propriétés (et notamment la transitivité) sont données à la section §3.6.

Dans les définitions qui suivent, nous parlons de « domaines de coût » sans démontrer que la relation définie est un préordre. Ce point est traité plus loin à la section §3.6, où sont données toutes les propriétés qui concernent ces coûts. On notera que certains nécessitent parfois une hypothèse technique supplémentaire que nous n'avons pas fait figurer explicitement dans les définitions.

3.4.1 Coût absolu.

Le coût absolu est la relation naturellement induite sur $\mathcal{C}^D$ par un coût dynamique $\preccurlyeq$ sur $\mathcal{C}$. La relation $p \preccurlyeq_{\text{abs}} q$ signifie que p a des coûts toujours inférieurs à ceux de q, quels que soient les paramètres.

Définition 3.19. (Coût absolu)
Le domaine de *coût absolu* $(\mathcal{C}^D, \preccurlyeq_{\mathrm{abs},D})$, déduit d'un coût dynamique $\preccurlyeq$ et d'un support D, est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\mathrm{abs},D} g \quad \text{ssi} \quad \forall d \in D \quad f(d) \preccurlyeq g(d)$

Dans le modèle $(\mathcal{C}^D, \preccurlyeq_{\mathrm{abs},D}, \mu)$, la relation sur les programmes se traduit par :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\mathrm{abs},D,\mu} q \quad \text{ssi} \quad \forall d \in D \quad \mu(p\ d) \preccurlyeq \mu(q\ d)$

Comme nous l'avons mentionné, on omet les indices D et μ lorsque le contexte n'est pas ambigu.

Le modèle résultant n'est pas total, comme en témoignent ces exemples dans BOOL.

```
fun nand1(X,Y) = if X then (if Y then false else true) else true
fun nand2(X,Y) = if Y then (if X then false else true) else true
fun nand3(X,Y) = if X then (if Y then false else X    ) else true
```

Les programmes `nand1`, `nand2` et `nand3`, tous équivalents entre eux, ont pour domaine BoolVal × BoolVal. Pour calculer leur coût dynamique, nous employons la formulation de la section §3.2.5 (implémentation informelle).

$$\mu(\texttt{nand1}\,(X,Y)) = \begin{cases} \text{si } X = \mathit{true} & \text{alors } 2\,c_{\texttt{if}} + 2\,c_{var} + c_{bool} \\ \text{si } X = \mathit{false} & \text{alors } c_{\texttt{if}} + c_{var} + c_{bool} \end{cases}$$

$$\mu(\texttt{nand2}\,(X,Y)) = \begin{cases} \text{si } Y = \mathit{true} & \text{alors } 2\ c_{\texttt{if}} + 2\ c_{var} + c_{bool} \\ \text{si } Y = \mathit{false} & \text{alors } c_{\texttt{if}} + c_{var} + c_{bool} \end{cases}$$

$$\mu(\texttt{nand3}\,(X,Y)) = \begin{cases} \text{si } X = \mathit{true} \text{ et } Y = \mathit{true} & \text{alors } 2\ c_{\texttt{if}} + 2\ c_{var} + c_{bool} \\ \text{si } X = \mathit{true} \text{ et } Y = \mathit{false} & \text{alors } 2\ c_{\texttt{if}} + 3\ c_{var} \\ \text{si } X = \mathit{false} & \text{alors } c_{\texttt{if}} + c_{var} + c_{bool} \end{cases}$$

Sachant que $c_{bool} \prec c_{var}$ (cf. §3.2.5, domaine de coût implicite), nous pouvons comparer `nand1` $\prec_{\mathrm{abs}}$ `nand3`. Par contre, Le programme `nand2` n'est comparable à aucun des deux autres car il est meilleur sur $(\mathit{true}, \mathit{false})$, mais moins bon sur $(\mathit{false}, \mathit{true})$. Donc `nand2` $\not\lesseqgtr_{\mathrm{abs}}$ `nand1` , `nand3`. Le coût absolu est peu nuancé ; il ne compare qu'un faible nombre de programmes car une seule donnée du domaine peut faire échouer la comparaison.

3.4.2 Coût presque partout.

Le coût *presque partout* est un procédé de construction de coût statique à partir d'un coût statique général $(\preccurlyeq_D)_{D \subset \mathcal{D}}$, qu'il *affaiblit* : la relation $p \preccurlyeq_{\mathrm{pp}} q$ signifie que p est meilleur que q au sens de $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ sauf, éventuellement, sur un nombre fini de données. Pour cela, il nous faut pouvoir restreindre l'expression d'un coût à un sous-ensemble du support. C'est l'objet du *coût restreint*.

Définition 3.20. (Coût restreint)
Soit $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ une famille de relations de coût. Pour tout support $D \subset \mathcal{D}$ et tout sous-ensemble $D' \subset D$, la relation $\preccurlyeq_{D|D'}$ de *coût de support D restreint à D'* est définie par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{D|D'} g \quad \text{ssi} \quad f_{|D'} \preccurlyeq_{D'} g_{|D'}$

Si D' est un sous-ensemble quelconque de $\mathcal{D}$, on note $\preccurlyeq_{D|D'}$ la relation de coût restreint $\preccurlyeq_{D|D\cap D'}$.

Le coût restreint $\preccurlyeq_{D|D'}$ est une relation sur $\mathcal{C}^D$. Il diffère en cela du coût $\preccurlyeq_{D\cap D'}$, qui est une relation sur $\mathcal{C}^{D\cap D'}$. Il n'a pas d'intérêt en soi ; en revanche, il est employé dans la construction d'autres coûts, comme ici du coût presque partout.

Définition 3.21. (Coût presque partout)
Le domaine de *coût presque partout* $(\mathcal{C}^D, \preccurlyeq_{\mathrm{pp},D})$, déduit d'un coût statique général $(\preccurlyeq_D)_{D\subset\mathcal{D}}$, est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\mathrm{pp},D} g \quad \text{ssi} \quad \exists\Delta \text{ fini} \subset D \quad f \preccurlyeq_{D|D\setminus\Delta} g$

Cette relation s'écrit aussi $\preccurlyeq_{\mathrm{pp},D} = \bigcup_{\Delta \text{ fini}\subset D} \preccurlyeq_{D|D\setminus\Delta}$. On note $\preccurlyeq_{\mathrm{pp}(\Delta),D} = \preccurlyeq_{D|D\setminus\Delta}$ pour Δ fini $\subset D$. Pour assurer la transitivité de $\preccurlyeq_{\mathrm{pp}}$, il faut que $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ soit compatible avec la restriction du support (cf. §3.5.3).

Ce coût n'a d'intérêt que sur des supports infinis[7]. Il n'est d'aucune utilité pour les supports finis car un choix de $\Delta = D$, alors fini, rend tous les programmes $\approx_{\mathrm{pp}}$-équivalents entre eux[8] :

- $\forall D \text{ fini} \subset \mathcal{D} \quad \preccurlyeq_{\mathrm{pp},D} = \approx_{\mathrm{triv},D}$

En ce qui concerne BOOL, langage de domaines finis (déf. 2.1), nous ne sommes donc guère avancés : `nand1` $\approx_{\mathrm{pp}}$ `nand2` $\approx_{\mathrm{pp}}$ `nand3`. Mais considérons les deux programmes SML équivalents suivants, qui implémentent la multiplication par deux sur le domaine infini des entiers naturels en représentation unaire.

```
fun double1 O = O
  | double1 (S n) = S (S (double1 n))

fun double2 n = let
  fun dbl O = n
    | dbl (S m) = S(dbl m)
in dbl n end
```

Sans donner le détail d'un modèle d'exécution ni d'un modèle de performance, on peut imaginer qu'il leur correspond des coûts scalaires respectifs de la forme $\mu(\texttt{double1}(n)) = 2nc_0 + c_1$ et $\mu(\texttt{double2}(n)) = nc_0 + c_2$ avec $c_1 \prec c_2$: le programme `double2` nécessite un peu plus de travail à l'initialisation (stockage de la valeur n, etc.), mais construit deux fois moins de termes que `double1`. Ces mesures vérifient la relation $\mu(\texttt{double2}(n)) \preccurlyeq \mu(\texttt{double1}(n))$ ssi $n \geqslant (c_2 - c_1)/c_0 = n_0$. Par conséquent, `double2` $\preccurlyeq_{\text{abs-pp}}$ `double1` : le programme `double1` est meilleur que `double2` sur les $\lfloor n_0 \rfloor$ premiers entiers, mais est moins bon sur tous les suivants.

3.4.3 Coût selon une distribution.

Avant de définir les coûts moyens, nous introduisons la notion de *distribution*, qui permet de pondérer une mesure de coût selon une répartition des données.

Définition 3.22. (Distribution)
Soit $D \subset \mathcal{D}$ un ensemble de données.

[7] Si l'on considère les limites physiques des machines, les domaines sont toujours finis mais totalement impropres à l'analyse théorique. On est dans la situation paradoxale où l'on peut préferer *presque partout* un programme moins bon sur les données pratiques (de petite taille) mais excellent sur les données de taille supérieure à la mémoire, impraticables. Le coût moyen souffre la même critique.

[8] Une variante du coût presque partout définie comme $\preccurlyeq_{\mathrm{pp},D}$ sur les supports infinis et comme $\preccurlyeq_{\mathrm{abs},D}$ sur les support finis éviterait ce problème.

- Une *distribution* sur D est une application $\nu : D \to \mathrm{IR}_+$
- Une *distribution normale* est une distribution $\nu : D \to [0,1]$ telle que $\sum_{d \in D} \nu(d) = 1$
- Une *distribution uniforme* est une distribution constante non nulle
- Une distribution ν est *strictement posivite*, noté $\nu > 0$, ssi $\nu : D \to \mathrm{IR}_+ \setminus \{0\}$
- Pour tout $f \in \mathcal{C}^D$, l'application $\nu.f \in \mathcal{C}^D$ est définie par : $\forall d \in D \quad (\nu.f)(d) = \nu(d)f(d)$

Nous notons $Dist(D)$ l'ensemble des distributions sur D.

Une distribution normale joue le rôle d'une probabilité sur D. Les distributions, comme les coûts, sont définis à une constante multiplicative près. Les poids $\nu(d)$ définissent simplement l'importance relative des données les unes par rapport aux autres.

Définition 3.23. (Coût selon une distribution)
Le *domaine* $(\mathcal{C}^D, \preccurlyeq_D^\nu)$ *de coût statique selon la distribution* ν, déduit du domaine $(\mathcal{C}^D, \preccurlyeq_D)$, est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_D^\nu g \quad \text{ssi} \quad \nu.f \preccurlyeq_D \nu.g$

Dans le modèle $(\mathcal{C}^D, \preccurlyeq_D^\nu, \mu)$, la relation $\preccurlyeq_D^\nu$ se traduit ainsi sur les programmes :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{D,\mu}^\nu q \quad \text{ssi} \quad \nu.\mu(p) \preccurlyeq_D \nu.\mu(q)$

La *mesure de coût* μ^ν *selon la distribution* ν est défini par :

- $\mu^\nu : \mathcal{P}_D \to \mathcal{C} \ \text{ tq } \ \forall p \in \mathcal{P}_D \quad \mu^\nu(p) = \nu.\mu(p)$

Les modèles $(\mathcal{C}^D, \preccurlyeq_D^\nu, \mu)$ et $(\mathcal{C}^D, \preccurlyeq_D, \mu^\nu)$ sont équivalents : on a $(\preccurlyeq_D^\nu)_\mu = (\preccurlyeq_D)_{\mu^\nu}$ sur $\mathcal{P}_D$.

Il est équivalent d'appliquer la distribution ν au coût statique $\preccurlyeq_D$ ou à la mesure μ. Si le coût statique de départ est noté $\preccurlyeq_{\mathrm{stat},D}$, il faut a priori différencier $\preccurlyeq_{\mathrm{stat},D}^\nu$, défini ci-dessus, de $(\preccurlyeq_D^\nu)_{\mathrm{stat},D}$.

L'influence de la répartition des données n'est pas une préoccupation artificielle. Par exemple, il est important de savoir qu'un tri comme `quicksort`, $\mathcal{O}(n \log n)$ en moyenne, ne convient pas pour une remise à jour après une petite modification car il est quadratique sur les listes « presque triées ». Cependant, le maniement pratique du coût selon une distribution est délicat, parce que l'on a en général qu'une faible connaissance de la répartition effective des données qui vont être fournies au programme. Il est assez rare de voir apparaître cette information dans la spécification ou l'interface d'un programme.

3.4.4 Coût moyen.

L'étude de la *complexité* d'un programme donne une mesure du coût d'exécution *moyen asymptotique* en fonction de la *taille* de ses arguments. Nous désirons dans cette section nous affranchir cette notion de taille, qui n'est pas *intrinsèque*. Nous examinons toutefois les coûts asymptotiques dans une section ultérieure (§3.4.9).

Les domaines de définition des programmes sont en règle générale des ensembles infinis[9]. Il est difficile de comparer un programme p à un programme q si les sommes de coûts $\mu(p) = \sum_{d \in D} \mu(p\ d)$ et $\mu(q) = \sum_{d \in D} \mu(q\ d)$ divergent. Toutefois, comme nous l'avons dit dans l'introduction de la section sur le coût statique

[9] Ils sont en pratique dénombrables (les réels, par exemple, n'en portent que le nom). D'autre part, le paradoxe mentionné à propos du coût presque partout (cf. §3.4.2, note de bas de page numéro 7) tient aussi : on peut être meilleur *en moyenne* sur les données inaccessibles en termes physiques. Si l'on a deux programmes p_1 et p_2 de complexité moyenne respective $\mu_1(n)$ et $\mu_2(n)$ en fonctions de la taille n de leurs arguments, bien que $\mu_1(n)$ puisse être asymptotiquement négligeable devant $\mu_2(n)$ en termes asymptotiques, il peut être prépondérant sur les données pratiques, par exemple à cause d'une constante de proportionnalité démesurément grande.

(cf. §3.3), ce qui nous importe est de comparer p à q, et non d'expliciter des valeurs $\mu(p)$ et $\mu(q)$. Il nous faut donc simplement connaître le signe de $\sum_{d\in D}(\mu(q\ d) - \mu(p\ d))$.

Cependant, on ne peut pas toujours donner un sens à une telle sommation. Par exemple, bien que la série $\sum_{n=1}^{\infty}(-1)^n/n$ converge, on peut donner n'importe quelle « limite » (et même la faire diverger) à la somme informelle $\sum_{n\geqslant 1}(-1)^n/n$, suivant la manière dont on regroupe les termes à sommer. Pour donner un exemple plus concret, considérons par exemple les deux programmes SML équivalents suivants, qui testent la parité des entiers relatifs.

```
fun pair1(x) = if (x mod 2)=0 then true       else not(true)
fun pair2(x) = if (x mod 2)=0 then not(false) else false
```

En supposant constant le calcul du modulo à deux (ou tout du moins indépendant de la parité), ces deux programmes, de domaine $\mathbb{Z}$, ont des coûts de la forme :

$$\begin{array}{rcl} \mu(\texttt{pair1}\,(x)) & = & \text{si } x \text{ est pair alors } c \text{ sinon } c + c_{\texttt{not}} \\ \mu(\texttt{pair2}\,(x)) & = & \text{si } x \text{ est pair alors } c + c_{\texttt{not}} \text{ sinon } c \end{array}$$

On est tenté de dire que ces programmes sont *équivalents en moyenne*. Considérons pourtant $(D_i)_{i\in\mathbb{Z}}$, la partition infinie en ensembles finis de leur domaine $\mathbb{Z}$, définie par $D_i = \{2i+1, 4i, 4i+2\}$. Nous avons pour tout $i \in \mathbb{Z}$ les coûts moyens ordinaires $\mu_{\text{moy}}(\texttt{pair1}_{|D_i}) = c + \frac{1}{3}\,c_{\texttt{not}}$ et $\mu_{\text{moy}}(\texttt{pair2}_{|D_i}) = c + \frac{2}{3}\,c_{\texttt{not}}$. Par conséquent, $\texttt{pair1} \prec_{\text{moy}|D_i} \texttt{pair2}$: le programme `pair1` est *strictement* meilleur en moyenne que `pair2` sur chaque élément de la partition $(D_i)_{i\in\mathbb{Z}}$. On ne peut pourtant en déduire que $\texttt{pair1} \prec_{\text{moy}} \texttt{pair2}$ car la partition symétrique $\{2i, 4i+1, 4i+3\}_{i\in\mathbb{Z}}$ conduirait à la contradiction $\texttt{pair2} \prec_{\text{moy}} \texttt{pair1}$. Un modèle qui « trancherait » finalement pour $\texttt{pair1} \approx_{\text{moy}} \texttt{pair2}$ comparerait énormément de programmes disparates. Nous préférons dire ici que `pair1` et `pair2` *ne sont pas* comparables en moyenne : $\texttt{pair1} \not\lessgtr_{\text{moy}} \texttt{pair2}$.

Ce problème correspond à la notion de *famille sommable* [RDO77] dans les espace vectoriel normés (resp. *somme partielle généralisée* pour les séries). Si la limite de la série $\sum_{n=1}^{\infty}(-1)^n/n$ dépend de la manière dont on somme les termes, c'est parce qu'elle ne converge pas *normalement*. En fait, dans le cas d'un espace de Banach, ce sont exactement les *familles absolument sommables* (resp. les *séries normalement convergentes*) qui sont indifférentes à l'ordre de sommation. La convergence de la somme des *modules* dans un treillis vectoriel (espace de Riesz) est également suffisante pour assurer l'indépendance de l'ordre de sommation.

En ce qui nous concerne, si $\sum_{d\in D}(\mu(q\ d) - \mu(p\ d))$ est sommable de somme c, la comparaison de p avec q est équivalente à la connaissance du signe de c. Mais nous pouvons également statuer sur cette comparaison si cette somme *diverge normalement avec un signe constant*.

Le problème ne se pose pas sur un support fini (c'est le cas en particulier d'un langage de domaines finis) car la sommation converge trivialement. De plus, l'ensemble $\mathcal{C}$ des coûts dynamiques suffit à expliciter la comparaison de coût moyen sur $\mathcal{P}$; il n'est pas indispensable d'employer l'ensemble fonctionnel $\mathcal{C}^D$.

Définition 3.24. (Coût moyen sur un support fini)
Le domaine $(\mathcal{C}^D, \preccurlyeq_{\text{moy},D})$ de *coût moyen (implicite) sur un support fini* D déduit du domaine de coût algébrique $(\mathcal{C}, +, ., \preccurlyeq)$ est :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\text{moy},D} g \quad \text{ssi} \quad \sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} g(d)$

Dans le modèle $(\mathcal{C}^D, \preccurlyeq_{\text{moy},D}, \mu)$, la relation $\preccurlyeq_{\text{moy},D}$ se traduit ainsi sur les programmes :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\text{moy},D,\mu} q \quad \text{ssi} \quad \sum_{d\in D} \mu(p\ d) \preccurlyeq \sum_{d\in D} \mu(q\ d)$

La *mesure* $\mu_{\text{moy},D}$ *de coût moyen (explicite) sur un support fini* D est définie par :

- $\mu_{\mathrm{moy},D} : \mathcal{P}_D \to \mathcal{C}$ tq $\forall p \in \mathcal{P}_D \quad \mu_{\mathrm{moy},D}(p) = \sum_{d\in D} \mu(p\ d)/|D|$

Par convention, cette somme est nulle sur un support D vide. On a alors $\preccurlyeq_{\mathrm{moy},\varnothing} \ = \approx_{\mathrm{triv}}$. Les modèles $(\mathcal{C}^D, \preccurlyeq_{\mathrm{moy},D}, \mu)$ et $(\mathcal{C}, \preccurlyeq, \mu_{\mathrm{moy},D})$ sont équivalents[10] : $\preccurlyeq_{\mathrm{moy},D,\mu} = \preccurlyeq_{\mu_{\mathrm{moy},D}}$ sur les programmes $\mathcal{P}_D$.

Il est équivalent de calculer le coût moyen explicitement par la mesure μ_{moy}, ou implicitement par la relation de coût $\preccurlyeq_{\mathrm{moy}}$. Notons qu'il n'est pas indispensable de diviser la somme $\sum_{d\in D} \mu(p\ d)$ par $|D|$ car on ne compare que des programme de même support, mais la valeur ainsi obtenue correspond à la notion ordinaire de moyenne.

Le langage BOOL est précisément de domaines finis. Reprenons les programmes `nand`i.

```
fun nand1(X,Y) = if X then (if Y then false else true) else true
fun nand2(X,Y) = if Y then (if X then false else true) else true
fun nand3(X,Y) = if X then (if Y then false else X   ) else true
```

Nous obtenons :

$$\begin{aligned}
\mu_{\mathrm{moy}}(\mathtt{nand1}) &= 3/2\ c_{\mathtt{if}} + 3/2\ c_{var} + c_{bool}\\
\mu_{\mathrm{moy}}(\mathtt{nand2}) &= 3/2\ c_{\mathtt{if}} + 3/2\ c_{var} + c_{bool}\\
\mu_{\mathrm{moy}}(\mathtt{nand3}) &= 3/2\ c_{\mathtt{if}} + 7/4\ c_{var} + 3/4\ c_{bool}
\end{aligned}$$

Par conséquent, `nand2` $\approx_{\mathrm{moy}}$ `nand1` $\prec_{\mathrm{moy}}$ `nand3`, grâce aux hypothèses $0 \prec c_{bool} \prec c_{var} \preccurlyeq c_{\mathtt{if}}$. Soit ν la distribution normale définie par :

$$\begin{aligned}
\nu(\mathit{true}, \mathit{true}) &= 1/6\\
\nu(\mathit{true}, \mathit{false}) &= 1/6\\
\nu(\mathit{false}, \mathit{true}) &= 1/3\\
\nu(\mathit{false}, \mathit{false}) &= 1/3
\end{aligned}$$

Elle signifie qu'il y a deux fois moins de « chances » que le premier paramètre X soit true plutôt que false. Nous pouvons calculer les coûts moyens selon ν :

$$\begin{aligned}
\mu^{\nu}_{\mathrm{moy}}(\mathtt{nand1}) &= 4/3\ c_{\mathtt{if}} + 4/3\ c_{var} + c_{bool}\\
\mu^{\nu}_{\mathrm{moy}}(\mathtt{nand2}) &= 5/3\ c_{\mathtt{if}} + 5/3\ c_{var} + c_{bool}\\
\mu^{\nu}_{\mathrm{moy}}(\mathtt{nand3}) &= 4/3\ c_{\mathtt{if}} + 3/2\ c_{var} + 5/6\ c_{bool}
\end{aligned}$$

La hiérarchie est modifiée : `nand1` $\prec^{\nu}_{\mathrm{moy}}$ `nand3` $\prec^{\nu}_{\mathrm{moy}}$ `nand2`.

3.4.5 Coût fini.

La relation précédente peut être étendue au cas d'un support D infini en limitant l'expression du coût à un ensemble fini et en imposant la comparaison de coût absolu sur le reste du domaine[11]. Ce *coût fini*, nécessitant une opération de restriction, est un procédé de construction de coût statique à partir d'un coût statique général. Pour garantir la transitivité, il fait deux hypothèses techniques, que nous ne formulerons qu'à la section suivante, une fois les coûts principaux présentés : stabilité (cf. §3.5.1) et compatibilité avec la fusion faible des supports (cf. §3.5.4).

[10] En toute rigueur, il n'y a équivalence que si $\preccurlyeq$ est homothétique.

[11] On peut aussi envisager d'imposer n'importe quel coût statique, autre que le coût absolu, du moment qu'il autorise des supports infinis.

Définition 3.25. (Coût fini)
Le *domaine de coût fini* $(\mathcal{C}^D, \preccurlyeq_{\text{fini},D})$, déduit d'un support D et d'un coût statique $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\text{fini},D} g$ ssi $\exists\Delta$ fini $\subset D \quad f \preccurlyeq_{D|\Delta} g$ et $f \preccurlyeq_{\text{abs},D|D\setminus\Delta} g$

Cette relation s'écrit aussi $\preccurlyeq_{\text{fini},D} = \bigcup_{\Delta \text{ fini}\subset D}(\preccurlyeq_{D|\Delta} \cap \preccurlyeq_{\text{abs},D|D\setminus\Delta})$. On note $\preccurlyeq_{\text{fini}(\Delta),D} = \preccurlyeq_{D|\Delta} \cap \preccurlyeq_{\text{abs},D|D\setminus\Delta}$ pour Δ fini $\subset D$.

En se basant sur le coût moyen sur un support fini $(\preccurlyeq_{\text{moy},D})_{D\subset\mathcal{D}}$, on peut ainsi former une relation de *coût moyen fini* $\preccurlyeq_{\text{moy-fini},D,\mu}$:

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\text{moy-fini}} q$ ssi $\exists\Delta$ fini $\subset D \quad \sum_{d\in\Delta} \mu(p\ d) \preccurlyeq \sum_{d\in\Delta} \mu(q\ d)$ et $\forall d \in D\setminus\Delta \quad f(d) \preccurlyeq g(d)$

Le coût fini permet non seulement de comparer des programmes de domaine infini, mais aussi des programmes de *coût non borné*. Par exemple, le coût moyen fini compare les programmes `double1` et `double2` rencontrés à propos du coût presque partout (cf. §3.4.2). En effet, $\sum_{n=0}^{m} 2nc_0 + c_1 \geqslant \sum_{n=0}^{m} nc_0 + c_2$ ssi $m \geqslant 2\,(c_2 - c_1)/c_0 = m_0$. Posons $\Delta = \{n \in \mathbb{N} \mid n \leqslant m_0\}$. Le programme `double2` est meilleur que `double1`, d'une part en moyenne sur l'ensemble fini Δ, et d'autre part pour tout entier à l'extérieur de Δ. Nous avons donc `double2` $\preccurlyeq_{\text{moy-fini}}$ `double1`.

3.4.6 Coût presque fini.

La limite d'une *famille sommable* est spécifiée à l'aide de la *base d'idéaux engendrés par les parties finies* de l'ensemble d'indices. Cependant, la convergence nous importe moins ici que le signe de la sommation. La définition du coût fini est adaptée en conséquence : au lieu d'imposer le coût absolu sur le reste du support (c'est-à-dire sur $D \setminus \Delta$), le *coût presque fini* requiert simplement d'être meilleur sur tous les surensembles finis d'un ensemble fini (c'est-à-dire sur Δ). C'est un procédé de construction de coût statique à partir d'un coût statique général.

Définition 3.26. (Coût presque fini)
Le domaine de *coût presque fini* $(\mathcal{C}^D, \preccurlyeq_{\text{pfini},D})$, déduit d'une relation de coût statique générale $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ et d'un support D, est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\text{pfini},D} g$ ssi $\exists\Delta$ fini $\subset D \quad \forall\Delta'$ fini $\supset \Delta \quad f \preccurlyeq_{D|\Delta'} g$

On note $\preccurlyeq_{\text{pfini}(\Delta),D} = \bigcap_{\Delta' \text{ fini}\supset\Delta} \preccurlyeq_{D|\Delta'}$ pour tout Δ fini $\subset D$. La relation de coût presque fini s'écrit alors $\preccurlyeq_{\text{pfini},D} = \bigcup_{\Delta \text{ fini}\subset D} \preccurlyeq_{\text{pfini}(\Delta),D}$.

Par exemple, la relation de *coût moyen presque fini* $\preccurlyeq_{\text{moy-pfini},D,\mu}$ se traduit sur $\mathcal{P}_D$ par :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\text{moy-pfini}} q$ ssi $\exists\Delta$ fini $\subset D \quad \forall\Delta'$ fini $\supset \Delta \quad \sum_{d\in\Delta'} \mu(p\ d) \preccurlyeq \sum_{d\in\Delta'} \mu(q\ d)$

Considérons les deux fonctions f et g de $(\mathbb{R}^{\mathbb{N}}, +, ., \leqslant)$ définies par : $f(n) = 1 + 1/2^n$ pour $n \geqslant 0$, et $g(0) = 3$, $g(n) = 1$ pour $n \geqslant 1$ (il n'y a pas dans la réalité de programme ayant des coûts infiniment petits ; il faut imaginer qu'une distribution a altéré les mesures de coût). Le coût moyen presque fini compare $f \preccurlyeq_{\text{moy-pfini}} g$, car toutes les sommes partielles de f sur les sur-ensembles de $\Delta = \{0\}$ restent inférieures à celles de g. Bien que la somme des coûts de $g - f$ soit infiniment proche de 0, elle reste strictement positive. Nous avons donc en réalité $f \prec_{\text{moy-pfini}} g$.

3.4.7 Coût limite.

Le *coût moyen limite* veut faire en sorte que les fonctions f et g de l'exemple précédent, dont la somme des différences de coûts « converge vers 0 », soient considérées équivalentes. Plus généralement, le *coût limite*

$f \preccurlyeq_{\lim,D} g$ signifie que la comparaison $f \preccurlyeq_D g$ est vraie à un coût arbitrairement petit près. Il nécessite une topologie sur $\mathcal{C}$.

Définition 3.27. (Domaine de coût normé)
Un *domaine de coût normé* $(\mathcal{C}, +, ., \preccurlyeq)$ est un domaine de coût algébrique tel que :

- $(\mathcal{C}, +, .)$ est un IR-espace vectoriel normé, complet
- La norme est *monotone* : $\forall c, c' \in \mathcal{C}$ si $0 \preccurlyeq c \preccurlyeq c'$ alors $0 \leqslant \|c\| \leqslant \|c'\|$

Si I est fini, l'espace vectoriel produit $(\prod_{i\in I} \mathcal{C}_i, +, .)$ de $(\mathcal{C}_i, +_i, ._i)_{i\in I}$ est normé[12] par :

- $\forall (c_i)_{i\in I} \in \prod_{i\in I} \mathcal{C}_i \quad \|(c_i)_{i\in I}\| = \sum_{i\in I} \|c_i\|_i$

L'espace vectoriel $(\mathcal{C}^D_{\text{norm}}, +, .)$ des fonctions f de $\mathcal{C}^D$ telles que $\sum_{d\in D} \|f(d)\|$ converge est normé par :

- $\forall f \in \mathcal{C}^D_{\text{norm}} \quad \|f\| = \sum_{d\in D} \|f(d)\|$

Pour tout $f \in \mathcal{C}^D_{\text{norm}}$, la somme $\sum_{d\in D} f(d)$ est définie car elle converge *normalement* (cf. §3.4.4).

La norme sur $\mathcal{C}^D_{\text{norm}}$ n'est pas nécessairement monotone pour toute relation $\preccurlyeq_D$. L'espace $(\mathcal{C}^D_{\text{norm}}, +, .)$ contient en particulier l'espace vectoriel des fonctions *presque nulles* (nulles sauf sur un ensemble fini).

Définition 3.28. (Coût limite)
Le domaine de *coût limite* $(\mathcal{C}^D, \preccurlyeq_{\lim,D})$, déduit d'un coût statique $\preccurlyeq_D$ additif et normé est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\lim,D} g$ ssi $\forall \varepsilon > 0 \ \ \exists h \in \mathcal{C}^D_{\text{norm}} \ \ \|h\| \leqslant \varepsilon$ et $f \preccurlyeq_D g + h$

L'additivité est indispensable pour la transitivité (tabl. 3.5).

Le cas particulier de *coût moyen limite* $f \preccurlyeq_{\text{moy-pfini-lim}} g$, en fait presque fini mais que nous abrégeons en $f \preccurlyeq_{\text{moy-lim}} g$, s'exprime ainsi :

- $\forall \varepsilon > 0 \ \ \exists c \in \mathcal{C} \ \ \|c\| \leqslant \varepsilon$ et $\exists \Delta$ fini $\subset D \ \ \forall \Delta'$ fini $\supset \Delta \quad \sum_{d\in\Delta'} f(d) \preccurlyeq \sum_{d\in\Delta'} g(d) + c$

La *convergence normale* nous permet de définir également :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\text{moy-cvn}} g$ ssi $g - f \in \mathcal{C}^D_{\text{norm}}$ et $0 \preccurlyeq \sum_{d\in D}(g - f)(d)$

La définition de $\preccurlyeq_{\text{moy-cvn}}$, *coût moyen normalement convergent*, est bien indépendante de l'ordre de sommation mais impose la convergence des sommes ; elle ne laisse pas la possibilité d'une divergence avec un signe constant. Elle est étendue au coût moyen limite par la proposition suivante, qui fournit aussi un moyen plus simple de décider si $f \preccurlyeq_{\text{moy-lim}} g$.

Proposition 3.1. (Condition suffisante pour le coût moyen limite)
Soit $(\mathcal{C}, +, ., \preccurlyeq, \sup, \inf)$ un domaine de coût additif et normé tel que :

- $(\mathcal{C}, \preccurlyeq, \sup, \inf)$ est un treillis (on note $|c| = \sup(c, 0) + \inf(c, 0)$)
- Le treillis est *normé*[13] : $\forall c, c' \in \mathcal{C}$ si $|c| \preccurlyeq |c'|$ alors $\|c\| \leqslant \|c'\|$

On a alors :

- $\forall f, g \in \mathcal{C}^D$ si $\exists h \in \mathcal{C}^D \ \ f \preccurlyeq_{\text{moy-cvn}} h \preccurlyeq_{\text{moy-pfini}} g$ alors $f \preccurlyeq_{\text{moy-pfini-lim}} g$

Il suffit en particulier que $f \preccurlyeq_{\text{moy-cvn}} h \preccurlyeq_{\text{abs}} g$. Symétriquement, $f \preccurlyeq_{\text{moy-pfini}} h \preccurlyeq_{\text{moy-cvn}} g$ est aussi une condition suffisante.

[12] Les normes définies par $\|(c_i)_{i\in I}\| = \max_{i\in I}(\|c_i\|_i)$ ou $\sqrt{\sum_{i\in I} \|c_i\|_i^2}$ définissent la même topologie. Il en va de même pour la norme (partielle) sur $\mathcal{C}^D$ définie ci-après.

[13] Puisque la norme est monotone par hypothèse (déf. 3.27), une condition équivalente est simplement $\forall c \in \mathcal{C} \quad \|\,|c|\,\| = \|c\|$.

Autrement dit, pour prouver $f \preccurlyeq_{\text{moy-lim}} g$, il suffit de trouver une fonction h partout dominée par g, et dont la différence avec f est absolument sommable, de somme positive. On a séparé d'une part la convergence de $h - f$, et d'autre part une majoration $h \preccurlyeq_{\text{moy-pfini}} g$ (une condition suffisante est $h \preccurlyeq_{\text{abs}} g$), qui permet à g d'être « infiniment » supérieur à h.

La relation $\preccurlyeq_{\text{moy-lim}}$ compare les fonctions f et g données en exemple du coût moyen presque fini (cf. §3.4.6). Alors que nous n'avions que $f \prec_{\text{moy-pfini}} g$, nous pouvons ici « passer à la limite » : $f \approx_{\text{moy-lim}} g$. Le coût moyen limite fait davantage que compléter certaines relations déjà existantes sur $\preccurlyeq_{\text{moy-pfini}}$. Considérons par exemple les fonctions f' et g' de $(\mathrm{IR}^{\mathrm{IN}}, +, ., \leqslant)$ définies par : $f'(2n) = 0$, $f'(2n+1) = 1/2^n$, et $g'(n) = f'(n+1)$, pour $n \in \mathrm{IN}$. Elles ne sont pas comparables par $\preccurlyeq_{\text{moy-pfini}}$, mais sont $\approx_{\text{moy-lim}}$-équivalentes.

Notons également que $\preccurlyeq_{\text{moy-lim}}$ ne compare effectivement pas `pair1` et `pair2`, dont la différence de coût ne converge pas normalement. En fait, un coût moyen qui affirmerait `pair1` $\approx_{\text{moy}}$ `pair2` déciderait aussi équivalents tous les programmes dont la différence de coût ne converge pas normalement.

3.4.8 Coût maximum.

Un algorithme est assez bien caractérisé par l'étude de son comportement en moyenne[14]. Cette analyse peut être complétée par l'étude du pire cas, c'est-à-dire la recherche du *coût maximum*. En pratique, cela n'est vraiment possible que sur un support fini car, si les coûts ne sont pas bornés (et c'est le cas le plus fréquent), il n'est pas possible de comparer deux programmes selon une notion de coût maximum. Comme pour le coût moyen, on peut toutefois étudier leur comportement asymptotique (cf. §3.4.9).

Définition 3.29. (Coût maximum)
Le domaine $(\mathcal{C}^D, \preccurlyeq_{\max,D})$ de *coût maximum (implicite)*, déduit d'un coût dynamique $\preccurlyeq$ et d'un support D, est :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\max,D} g \quad \text{ssi} \quad \exists d' \in D \quad \forall d \in D \quad f(d) \preccurlyeq g(d')$

Dans le modèle $(\mathcal{C}^D, \preccurlyeq_{\max,D}, \mu)$, la relation $\preccurlyeq_{\max,D}$ se traduit ainsi sur les programmes :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\max,D,\mu} q \quad \text{ssi} \quad \exists d' \in D \quad \forall d \in D \quad \mu(p\ d) \preccurlyeq \mu(q\ d')$

Si $\preccurlyeq$ est totale, la *mesure* $\mu_{\max}$ *de coût maximum (explicite) sur un support D fini non-vide* est définie par :

- $\forall C$ fini, non-vide $\subset \mathcal{C} \quad \max(C) = \max_{c \in C}(c) = c'$ tel que $c' \in C$ et $\forall c \in C \quad c \preccurlyeq c'$
- $\mu_{\max,D} : \mathcal{P}_D \to \mathcal{C}$ tq $\forall p \in \mathcal{P}_D \quad \mu_{\max}(p) = \max(\mu(p\ D)) = \max_{d \in D}(\mu(p\ d))$

La *mesure de coût maximum* $\mu_{\max}$ se traduit ainsi en terme de comparaison de programmes :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\max} q \quad \text{ssi} \quad \max_{d \in D} \mu(p\ d) \preccurlyeq \max_{d \in D} \mu(q\ d)$

Les modèles $(\mathcal{C}^D, \preccurlyeq_{\max,D}, \mu)$ et $(\mathcal{C}, \preccurlyeq, \mu_{\max,D})$ sont équivalent : on a $\preccurlyeq_{\max,D,\mu} = \preccurlyeq_{\mu_{\max,D}}$ sur $\mathcal{P}_D$.

Il est équivalent de calculer le coût maximum explicitement par la mesure $\mu_{\max}$, ou implicitement par la relation de coût $\preccurlyeq_{\max}$.

Dans l'optique d'une optimisation, la comparaison de coût $\preccurlyeq_{\max}$ a, seule, relativement peu d'intérêt. En revanche, elle convient bien pour renforcer une autre comparaison, comme par exemple $\preccurlyeq_{\text{moy-max}} = \preccurlyeq_{\text{moy}} \cap \preccurlyeq_{\max}$, que l'on a mentionné à propos du coût conjoint (cf. §3.1.3). Par exemple, les programmes `nand`i de BOOL ont les coûts maximums suivants :

$$\mu_{\max}(\texttt{nand1}) \quad = \quad 2\,c_{\texttt{if}} + 2\,c_{var} + c_{bool}$$

[14] Pour mieux cerner le comportement en moyenne s'ajoute aussi l'étude de l'*écart type*.

$$\begin{aligned}\mu_{\max}(\texttt{nand2}) &= 2\,c_{\texttt{if}} + 2\,c_{var} + c_{bool}\\ \mu_{\max}(\texttt{nand3}) &= 2\,c_{\texttt{if}} + 3\,c_{var}\end{aligned}$$

Par conséquent, `nand2` $\approx_{\max}$ `nand1` $\prec_{\max}$ `nand3`. C'est la même hiérarchie que pour le coût moyen. Nous avons donc `nand2` $\approx_{\text{moy-max}}$ `nand1` $\prec_{\text{moy-max}}$ `nand3`.

Mais considérons maintenant, toujours dans BOOL, les deux programmes `foo1` et `foo2` équivalents[15] suivants.

```
fun foo1(X,Y,Z) = if X then if Y then true else Z
                       else if Z then Y    else true

fun foo2(X,Y,Z) = if Y then true
                       else if X then Z
                                 else if Z then false else true
```

Ils ont pour coût moyen :

$$\begin{aligned}\mu_{\text{moy}}(\texttt{foo1}) &= 2\ c_{\texttt{if}} + 5/2\ c_{var} + 1/2\ c_{bool}\\ \mu_{\text{moy}}(\texttt{foo2}) &= 7/4\ c_{\texttt{if}} + 2\ c_{var} + 3/4\ c_{bool}\end{aligned}$$

Grâce aux hypothèses $0 \prec c_{bool} \prec c_{var} \preccurlyeq c_{\texttt{if}}$, nous obtenons `foo1` $\succ_{\text{moy}}$ `foo2` : le programme `foo2` est préférable en moyenne à `foo1`. Ces mêmes hypothèses nous permettent de calculer les coûts maximaux

$$\begin{aligned}\mu_{\max}(\texttt{foo1}) &= 2\ c_{\texttt{if}} + 3\ c_{var}\\ \mu_{\max}(\texttt{foo2}) &= 3\ c_{\texttt{if}} + 3\ c_{var} + c_{bool}\end{aligned}$$

et de comparer `foo1` $\prec_{\max}$ `foo2`. Au total, il est peut être avantageux en moyenne de remplacer `foo1` par `foo2`, mais il faut savoir que le pire cas de performance est encore plus mauvais : `foo1` $\not\lessgtr_{\text{moy-max}}$ `foo2`.

3.4.9 Coûts asymptotiques.

Nous nous sommes efforcés de donner une définition du coût moyen qui soit intrinsèque (cf. §3.4.4). L'obstacle majeur était la dépendance dans l'ordre de sommation des coûts. Lorsqu'il existe une *énumération des données* qui fait un sens pour le problème posé, elle impose un ordre et donne une signification à toute somme. Introduisons tout d'abord la notion de *taille*.

Définition 3.30. (Taille)
Une *taille* sur un ensemble de données $D \subset \mathcal{D}$ est une application de $D \to \mathbb{N}$, notée $|\,.\,|$, qui associe à toute donnée d un entier, noté $|d|$, tel que les ensembles $\{d \in D \mid n = |d|\}$ soient finis pour tout $n \in \mathbb{N}$.

C'est la définition adoptée dans les formulations usuelles de complexité asymptotique. Il faut quotienter l'ensemble des données par une relation d'équivalence si l'on veut pouvoir définir une fonction taille qui, par exemple, associe à une liste sa longueur[16] Notons aussi que :

- $\{\{d \in D \mid n = |d|\} \mid n \in \mathbb{N}\}$ est une partition de D
- $\{d \in D \mid n \leqslant |d|\}$ est fini pour tout $n \in \mathbb{N}$

[15] Tous deux implémentent $Y \vee (X \Leftrightarrow Z)$.

[16] Dans ce cas, le programme que l'on désire analyser ne doit pas aller examiner le contenu des cellules de liste.

- $\bigcup_{n \in \mathbb{N}} \{d \in D \mid n \leqslant |d|\} = \bigcup_{n \in \mathbb{N}} \{d \in D \mid n = |d|\} = D$

Les coûts asymptotiques sont des procédés de construction de coût statique à partir d'un coût statique général de supports *finis*. Ces supports sont les ensembles $\{d \in D \mid n = |d|\}$ pour le coût *asymptotique local*, qui étudie le comportement à l'infini des données de *même* taille, et $\{d \in D \mid n \leqslant |d|\}$ pour le coût *asymptotique global*, qui cumule l'analyse du comportement sur les données d'*au plus* une certaine taille.

Définition 3.31. (Coût asymptotique local)
Le domaine de *coût asymptotique local* $(\mathcal{C}^D, \preccurlyeq_{\text{asymloc},D})$, déduit d'un coût statique général de supports finis $(\preccurlyeq_D)_{D \text{ fini} \subset \mathcal{D}}$ et d'un support D, est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\text{asymloc},D} g \quad \text{ssi} \quad \exists N \in \mathbb{N} \quad \forall n \geqslant N \quad f \preccurlyeq_{D|\{d \in D \,:\, n=|d|\}} g$

On note $\preccurlyeq_{\text{asymloc}(N)} = \bigcap_{n \geqslant N} \preccurlyeq_{D|\{d \in D \,:\, n=|d|\}}$. La relation s'écrit alors $\preccurlyeq_{\text{asymloc}} = \bigcup_{N \in \mathbb{N}} \preccurlyeq_{\text{asymloc}(N)}$.

Définition 3.32. (Coût asymptotique global)
Le domaine de *coût asymptotique global* $(\mathcal{C}^D, \preccurlyeq_{\text{asymglob},D})$, déduit d'un coût statique général de supports finis $(\preccurlyeq_D)_{D \text{ fini} \subset \mathcal{D}}$ et d'un support D, est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\text{asymglob},D} g \quad \text{ssi} \quad \exists N \in \mathbb{N} \quad \forall n \geqslant N \quad f \preccurlyeq_{D|\{d \in D \,:\, n \geqslant |d|\}} g$

On note $\preccurlyeq_{\text{asymglob}(N)} = \bigcap_{n \geqslant N} \preccurlyeq_{D|\{d \in D \,:\, n \geqslant |d|\}}$. La relation s'écrit alors $\preccurlyeq_{\text{asymglob}} = \bigcup_{N \in \mathbb{N}} \preccurlyeq_{\text{asymglob}(N)}$.

Nous avons ainsi :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\text{moy-asymloc}} q \quad \text{ssi} \quad \exists N \in \mathbb{N} \quad \forall n \geqslant N \quad \sum_{|d|=n} \mu(p\ d) \preccurlyeq \sum_{|d|=n} \mu(q\ d)$
- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\text{max-asymglob}} q \quad \text{ssi} \quad \exists N \in \mathbb{N} \quad \forall n \geqslant N \quad \max_{|d| \leqslant n} \mu(p\ d) \preccurlyeq \max_{|d| \leqslant n} \mu(q\ d)$

Considérons par exemple le cas où $D = \mathbb{N}^*$, et où la taille $|d|$ d'un entier $d \in \mathbb{N}^*$ est le nombre de symboles dans sa représentation en base 2. Autrement dit,

- $\forall d \in \mathbb{N}^* \quad |d| = \lfloor \log_2(d) \rfloor + 1$

Réciproquement,

- $|d| = n \quad \text{ssi} \quad d \in \{2^{n-1}, \ldots, 2^n - 1\}$

On a alors :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\text{moy-asymloc}} q \quad \text{ssi} \quad \exists N \in \mathbb{N} \quad \forall n \geqslant N \quad \sum_{d=2^{n-1}}^{2^n-1} \mu(p\ d) \preccurlyeq \sum_{d=2^{n-1}}^{2^n-1} \mu(q\ d)$
- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\text{max-asymglob}} q \quad \text{ssi} \quad \exists N \in \mathbb{N} \quad \forall n \geqslant N \quad \exists d < 2^n \quad \forall d' < 2^n \quad \mu(p\ d') \preccurlyeq \mu(q\ d')$

Dans ce dernier cas, nous avons supposé que la relation $\preccurlyeq$ était totale.

Il faut noter que le coût asymptotique local et le coût asymptotique global sont en général différents. La section §3.6.6 donne quelques relations entre eux.

3.4.10 Coûts majorés.

L'analyse de la complexité d'un programme donne une image fine du comportement en moyenne : elle dit qu'un algorithme est *linéaire*, *quadratique*, etc. (en fonctions de la tailles des données). L'étude du *rapport de performance* de deux programmes donne une appréciation de la complexité moins précise, mais en revanche plus simple et plus systématique. C'est l'approche employée par exemple par Andersen et Gomard [AG92] pour analyser la performance de la spécialisation (cf. §1.3.4, objet des transformations). Elle tranche avec les coûts précédents où nous avons chiffré en quelque sorte la *différence de performance*, qui ne permet pas de distinguer une optimisation constante d'un gain de nature exponentielle.

Un programme p est inférieur à un programme q selon le *coût majoré* si le coût de p est inférieur à celui de q d'un facteur linéaire au moins $\alpha \in \mathrm{IR}_+$.

- Le cas $\alpha > 1$ donne un sens au remplacement de q par p dans le but d'optimiser.
- Le cas $\alpha = 1$ n'altère pas la comparaison de départ.
- Le cas $0 < \alpha < 1$ correspond à la relation réciproque du cas $1/\alpha > 1$.
- Le cas $\alpha = 0$ exprime que q est positif ; il ne dit rien sur p.

C'est bien entendu le cas $\alpha > 1$ qui nous intéresse, mais les autres valeurs de α peuvent aussi intervenir de manière indirecte.

Définition 3.33. (Coût majoré, coût négligeable)
Soient $(\mathcal{C}^D, +, ., \preccurlyeq_D)$ un domaine de coût statique algébrique et $A \subset \mathrm{IR}_+$. Les définitions des comparaisons de coût suivantes s'entendent pour tout $f, g \in \mathcal{C}^D$.

- $f \preccurlyeq_{\mathrm{maj}(A),D} g$ ssi $\forall \alpha \in A \quad \alpha.f \preccurlyeq_D g$ définit le *coût majoré par des facteurs A*
- $f \preccurlyeq_{\mathrm{maj}(\alpha),D} g$ ssi $\forall \alpha' \in [0, \alpha] \quad \alpha'.f \preccurlyeq_D g$ définit le *coût majoré d'un facteur au moins $\alpha > 1$*
- $f \preccurlyeq_{\mathrm{majl}(\alpha),D} g$ ssi $\forall \alpha' \in [0, \alpha[\quad \alpha'.f \preccurlyeq_D g$ définit le *coût majoré d'un facteur limite au moins $\alpha > 1$*
- $f \preccurlyeq_{\mathrm{maj},D} g$ ssi $\exists \alpha \in \mathrm{IR},\, 1 < \alpha \quad f \preccurlyeq_{\mathrm{maj}(\alpha),D} g$ définit le *coût majoré*
- $f \preccurlyeq_{\text{négl},D} g$ ssi $\forall \alpha \in \mathrm{IR}_+ \quad \alpha.f \preccurlyeq_D g$ définit le *coût négligeable*

Ces relations s'écrivent aussi :

- $\preccurlyeq_{\mathrm{maj}(\alpha)} = \preccurlyeq_{\mathrm{maj}([0,\alpha])} = \bigcap_{0 \leqslant \alpha' \leqslant \alpha} \preccurlyeq_{\mathrm{maj}\{\alpha'\}}$
- $\preccurlyeq_{\mathrm{majl}(\alpha)} = \preccurlyeq_{\mathrm{maj}([0,\alpha[)} = \bigcap_{0 \leqslant \alpha' < \alpha} \preccurlyeq_{\mathrm{maj}\{\alpha'\}}$
- $\preccurlyeq_{\mathrm{maj}} = \bigcup_{1 < \alpha} \preccurlyeq_{\mathrm{maj}(\alpha)}$
- $\preccurlyeq_{\text{négl}} = \preccurlyeq_{\mathrm{maj}([0,+\infty[)} = \bigcap_{0 \leqslant \alpha} \preccurlyeq_{\mathrm{maj}\{\alpha\}}$

Ces notions sont similaires à celles de relations de domination et de prépondérance sur les fonctions de domaine réel dans les espaces vectoriels normés.

Les relations de coût majoré ne sont généralement pas additives : si l'on réduit d'un facteur $\alpha > 1$ le coût d'un programme p, on ne réduit pas d'un même facteur α le coût d'un programme plus large qui employerait p comme procédure. Elles ne sont généralement pas réflexives non plus car elles ne comparent pas à eux-mêmes les coûts strictement positifs. Pourtant, ce ne sont pas non plus des relations strictes (irréflexives) car elles comparent à eux-mêmes les coûts négatifs ou nuls. Puisqu'elles ne sont pas réflexives, elles ne sont pas totales non plus.

Reprenons l'exemple des programmes `double1` et `double2` rencontrés à propos des coûts presque partout (cf. §3.4.2) et moyen fini (cf. §3.4.5). Nous avions $\mu(\texttt{double1}\,(n)) = 2nc_0 + c_1$ et $\mu(\texttt{double2}\,(n)) = nc_0 + c_2$. Soit $0 \leqslant \alpha < 2$, alors $\sum_{n=0}^{m} 2nc_0 + c_1 \geqslant \sum_{n=0}^{m} \alpha(nc_0 + c_2)$ ssi $(2-\alpha)c_0\, m^2 + ((2-\alpha)c_0 + 2c_1 - \alpha c_2)\, m + 2c_1 - \alpha c_2 \geqslant 0$. Cette quantité est strictement positive à partir d'un certain rang m_0. Posons $\Delta = \{n \in \mathrm{IN} \mid n \leqslant m_0\}$. Alors `double1` $\succ_{\text{moy-fini-maj}(\alpha)}$ `double2` selon Δ. On en déduit `double1` $\succ_{\text{moy-fini-majl}(2)}$ `double2` et `double1` $\succ_{\text{moy-fini-maj}}$ `double2`. En revanche, `double1` $\not\succcurlyeq_{\text{moy-fini-maj}(2)}$ `double2` car $c_1 \prec c_2$.

Cette majoration est « finie ». Nous avons ici `double2` $\not\succcurlyeq_{\text{moy-fini-négl}}$ `double1`. En revanche, si nous reprenons les fonctions `f` et `f'` de la section §1.3.4 (objet des transformations),

```
fun g (n) = n + n
fun f (n) = if n = 0 then 1 else g(f(n-1))
fun f'(n) = if n = 0 then 1 else f'(n-1) + f'(n-1)
```

avec des coûts vraisemblables de la forme $\mu(\texttt{f}\,(n)) = nc_1 + c_2$ et $\mu(\texttt{f'}\,(n)) = 2^n c'_1 + c'_2$, nous obtenons $\texttt{f} \preccurlyeq_{\text{moy-fini-maj}(\alpha)} \texttt{f'}$ pour tout réel $\alpha \geqslant 0$, donc $\texttt{f} \preccurlyeq_{\text{moy-fini-négl}} \texttt{f'}$. Nous avons une comparaison similaire des versions linéaire et exponentielle de la fonction de Fibonaci.

3.5 Qualités d'une relation de coût statique.

Rien ne nous permet de décider a priori d'une *meilleure* comparaison de coût statique. Chacune reflète une caractéristique particulière des programmes.

Pour un coût statique $\preccurlyeq$ donné, déduit d'un ensemble de coûts statiques ou dynamiques $\preccurlyeq_i$, il faut néanmoins systématiquement s'assurer de la *transitivité*, sans laquelle il n'y a pas de « domaine de coût ». Nous examinons également la *réfléxivité*, l'*antisymétrie*, la *totalité* et l'*additivité*, sous l'hypothèse que la *même* propriété est vérifiée par chacun des coûts $\preccurlyeq_i$.

À l'instar des notations adoptées pour la conjonction et la disjonction des relations de coûts, et pour simplifier certaines démonstrations, nous identifions toute relation $\preccurlyeq$ avec son graphe, sous-ensemble de $\mathcal{C} \times \mathcal{C}$. Soit $(\mathcal{C}, \preccurlyeq_i)_{i\in I}$ une famille d'ensembles munis d'une relation, nous notons ainsi :

- $\preccurlyeq_1 \cdot \preccurlyeq_2 = \{(c,c') \in \mathcal{C}^2 \mid \exists c'' \in \mathcal{C} \quad c \preccurlyeq_1 c'' \preccurlyeq_2 c'\}$
- $\preccurlyeq_1 + \preccurlyeq_2 = \{(c_1 + c_2, c'_1 + c'_2) \in \mathcal{C}^2 \mid c_1 \preccurlyeq_1 c'_1 \ \text{ et } \ c_2 \preccurlyeq c'_2\}$
- $\alpha.\preccurlyeq = \{(\alpha.c, \alpha.c') \in \mathcal{C}^2 \mid c \preccurlyeq c'\}$ pour $\alpha \in \mathrm{IR}$
- $\preccurlyeq_1 \subset \preccurlyeq_2$ ssi $\forall c, c' \in \mathcal{C}$ si $c \preccurlyeq_1 c'$ alors $c \preccurlyeq_2 c'$
- $\bigcap_{i\in I} \preccurlyeq_i = \{(c,c') \in \mathcal{C}^2 \mid \forall i \in I \quad c \preccurlyeq_i c'\}$
- $\bigcup_{i\in I} \preccurlyeq_i = \{(c,c') \in \mathcal{C}^2 \mid \exists i \in I \quad c \preccurlyeq_i c'\}$
- $\not\preccurlyeq = \complement_{\mathcal{C}^2}(\preccurlyeq) = \mathcal{C}^2 \setminus \preccurlyeq$ et $\prec = \preccurlyeq \cap \not\succcurlyeq$ et $\approx = \preccurlyeq \cap \succcurlyeq$
- $\nabla_{\mathcal{C}} = \{(c,c) \in \mathcal{C}^2 \mid c \in \mathcal{C}\}$ est la *diagonale* de $\mathcal{C} \times \mathcal{C}$, c'est-à-dire la relation d'égalité « $=$ »
- $\approx_{\text{triv}} = \mathcal{C}^2$

Soit $(\mathcal{C}_i, \preccurlyeq_i)_{i\in I}$ une famille d'ensembles munis d'une relation.

- $\bigotimes_{i\in I} \preccurlyeq_i = \{((c_i)_{i\in I}, (c'_i)_{i\in I}) \in (\prod_{i\in I} \mathcal{C}_i)^2 \mid \forall i \in I \quad c_i \preccurlyeq_i c'_i\}$
- $\bigotimes^{\text{lex}}_{i\in I} \preccurlyeq_i = \bigcup_{i\in I}((\bigotimes_{j\in I, j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I, j>i} \approx_{j,\text{triv}})) \cup (\bigotimes_{i\in I} \preccurlyeq_i)$

On omet l'indice de ∇ et $\complement$ lorsqu'il n'y a pas ambiguïté sur l'ensemble de coûts. On a d'ailleurs :

- $\bigotimes_{i\in I} \nabla_{\mathcal{C}_i} = \nabla_{\prod_{i\in I} \mathcal{C}_i}$

Les propriétés élémentaires des relations de coûts se traduisent ainsi en termes de graphes :

- $\preccurlyeq$ est totale ssi $\preccurlyeq \cup \succcurlyeq = \mathcal{C}^2$
- $\preccurlyeq$ est refléxive ssi $\nabla \subset \preccurlyeq$
- $\preccurlyeq$ est symétrique ssi $\preccurlyeq = \succcurlyeq$
- $\preccurlyeq$ est antisymétrique ssi $\preccurlyeq \cap \succcurlyeq \subset \nabla$
- $\preccurlyeq$ est transitive ssi $\preccurlyeq \cdot \preccurlyeq \subset \preccurlyeq$
- $\preccurlyeq$ est additive ssi $\preccurlyeq + \nabla \subset \preccurlyeq$
- $\preccurlyeq$ est homothétique ssi $\forall \alpha > 0 \quad \alpha.\preccurlyeq \subset \preccurlyeq$

Les propriétés supplémentaires qui suivent sont spécifiques aux procédés de construction de coût statique.

3.5.1 Stabilité.

La propriété de *stabilité* exprime un lien de *cohérence* entre un coût dynamique et un coût statique déduit : si un programme p est meilleur que q à tout point de vue (sur chacune des données du support), il est naturel de penser que $p \preccurlyeq_{\mathrm{stat}} q$. Une autre manière de voir la chose est d'imaginer que si l'on améliore le comportement d'un programme sur quelques unes de ses données (donc au sens du coût absolu), on souhaite que le nouveau programme soit meilleur que l'ancien également au sens de $\preccurlyeq_{\mathrm{stat}}$. La relation de coût est « *stable* par amélioration locale ».

Définition 3.34. (Coût stable (ou cohérent avec le coût absolu))
Un coût statique $\preccurlyeq_{\mathrm{stat}}$ sur $\mathcal{C}^D$, déduit d'un coût dynamique $\preccurlyeq$, est *stable* (ou *cohérent avec le coût absolu*) ssi :

- $\forall f, g \in \mathcal{C}^D$ si $f \preccurlyeq_{\mathrm{abs}} g$ alors $f \preccurlyeq_{\mathrm{stat}} g$

Cette condition s'écrit encore : $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\mathrm{stat}}$.

3.5.2 Discrimination.

Si l'on améliore *strictement* le coût dynamique d'un programme p sur quelques unes de ses donnés, formant ainsi un nouveau programme $q \prec_{\mathrm{abs}} p$, il est souhaitable que la comparaison $\preccurlyeq_{\mathrm{stat}}$ reflète également cette amélioration stricte et en particulier vérifie $q \not\succcurlyeq_{\mathrm{stat}} p$.

Par exemple, le programme `nand1` est strictement meilleur que `nand3` pour le coût absolu (cf. §3.4.1) : `nand1` $\prec_{\mathrm{abs}}$ `nand3`. Il est également meilleur pour le coût moyen (cf. §3.4.4) : `nand1` $\prec_{\mathrm{moy}}$ `nand3`. En particulier, on a donc la relation `nand1` $\not\succcurlyeq_{\mathrm{moy}}$ `nand3`.

Cette condition n'est pas nécessairement vérifiée. Dans le cas où elle ne l'est pas, cela signifie que l'on peut substituer p à q dans une optimisation, c'est-à-dire remplacer q par un programme p strictement moins bon au sens du coût absolu. Une telle comparaison de coût $\preccurlyeq_{\mathrm{stat}}$ est peu intéressante car peu *discriminante* ; elle met en relation des programmes de comportement trop dissemblable.

Définition 3.35. (Coût discriminant)
Un coût statique $\preccurlyeq_{\mathrm{stat}}$ sur $\mathcal{C}^D$, déduit d'un coût dynamique $\preccurlyeq$, est *discriminant* ssi il vérifie l'une des propositions équivalentes suivantes.

- $\forall f, g \in \mathcal{C}^D$ si $f \prec_{\mathrm{abs}} g$ alors $f \not\succcurlyeq_{\mathrm{stat}} g$
- $\forall f, g \in \mathcal{C}^D$ si $f \preccurlyeq_{\mathrm{abs}} g \preccurlyeq_{\mathrm{stat}} f$ alors $f \approx_{\mathrm{abs}} g$

La condition s'écrit aussi $\preccurlyeq_{\mathrm{stat}} \subset \not\succ_{\mathrm{abs}}$, c'est-à-dire $\preccurlyeq_{\mathrm{stat}} \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$.

Les propriétés de stabilité et de discrimination sont indépendantes. Elles fournissent un encadrement « raisonnable » d'un coût statique : $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\mathrm{stat}} \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$. De plus, un coût statique stable et discriminant vérifie aussi (prop. 3.3) $\prec_{\mathrm{abs}} \subset \prec_{\mathrm{stat}}$, c'est-à-dire :

- $\forall f, g \in \mathcal{C}^D$ si $f \prec_{\mathrm{abs}} g$ alors $f \prec_{\mathrm{stat}} g$

Mais ce n'est pas une condition suffisante pour que $\preccurlyeq_{\mathrm{stat}}$ soit stable et discriminant.

Une relation de coût n'est pas discriminante lorsqu'il existe des données du domaine qui ne sont pas prises en considération dans une comparaison. C'est le cas par exemple du coût presque partout et du coût asymptotique local (cf. §3.6.4). C'est le cas également lorsque l'on peut « repousser à l'infini » la prise en compte de ces données, comme le montre le *coût permuté* défini ci-après.

Coût permuté.

Le *coût permuté* que nous définissons maintenant n'a pas d'intérêt en soi ; il a pour seul but de faire apparaître en quoi un coût qui n'est pas discriminant est peu désirable. Il dit qu'un programme p est meilleur qu'un programme q si l'on peut mettre en correspondance les coûts $(\mu(p\ d))_{d\in D}$ et $(\mu(q\ d))_{d\in D}$ de sorte que ceux de p soient toujours inférieurs à ceux de q. Autrement dit, on peut mettre côte à côte deux énumérations des coûts de p et q telles que chaque coût individuel pour p soit inférieur à son vis-à-vis pour q. C'est le coût absolu (qui est discriminant car $\preccurlyeq_{\mathrm{abs}} \subset \prec_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$) à une bijection près. Le domaine de *coût permuté* $(\mathcal{C}^D, \preccurlyeq_{\mathrm{perm},D})$ est défini par :

- $\forall f, g \in \mathcal{C}^D \quad f \preccurlyeq_{\mathrm{perm},D} g \quad \text{ssi} \quad \exists \beta \text{ bijection} \in D^D \quad \forall d \in D \quad f(\beta(d)) \preccurlyeq g(d)$

C'est un domaine (la relation $\preccurlyeq_{\mathrm{perm},D}$ est bien transitive) réflexif et stable, mais il n'est ni additif, ni discriminant.

Soit par exemple $(\mathcal{C}, \preccurlyeq) = (\mathbb{N}, \leqslant)$, $D = \{0,1\}$ et f, g les fonctions de $\mathbb{N}^{\{0,1\}}$ définie par $f = \langle 1, 3\rangle$ et $g = \langle 3, 1\rangle$. Nous avons $f \approx_{\mathrm{perm}} g$, pourtant $\langle 2, 6\rangle = f + f \not\preccurlyeq_{\mathrm{perm}} g + f = \langle 4, 4\rangle$. La relation $\preccurlyeq_{\mathrm{perm}}$ n'est donc pas additive.

La relation n'est pas non plus discriminante. Soit f et h les fonctions de $(\mathbb{N}^{\mathbb{N}}, +, ., \leqslant)$ définies par : $f(n) = n$ si n est impair et 0 sinon, $h(n) = 1$ si n est impair et 0 sinon. Autrement dit, $f = \langle 0, 1, 0, 3, 0, 5, \ldots\rangle$ et $h = \langle 0, 1, 0, 1, 0, 1, \ldots\rangle$. Posons $g = f + h = \langle 0, 2, 0, 4, 0, 6, \ldots\rangle$. Nous avons alors $f \preccurlyeq_{\mathrm{abs}} g \preccurlyeq_{\mathrm{perm}} f$. Pourtant $g \not\preccurlyeq_{\mathrm{abs}} f$. En fait, nous avons plus généralement pour tout $n \in \mathbb{N}$, $f + nh \preccurlyeq_{\mathrm{perm}} f$: on préfère à f une fonction $f + nh$ de coût « arbitrairement » plus mauvais au sens du coût absolu!

Le coût permuté permet également de comparer les programmes `pair1` et `pair2` rencontrés à propos du coût moyen (cf. §3.4.4), que nous avions convenu de ne pas mettre en relation pour échapper à une incohérence : `pair1` $\approx_{\mathrm{perm}}$ `pair2`. Ces exemples illustrent à quel point le caractère discriminant est important.

3.5.3 Compatibilité avec la restriction du support.

La *compatibilité avec la restriction du support* signifie que si un programme p est meilleur que q sur un ensemble de données D (au sens d'un coût statique donné), alors il est aussi meilleur sur tous les sous-ensembles de D.

Définition 3.36. (Coût compatible avec la restriction (du support))
Un coût statique $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est *compatible avec la restriction (du support)* ssi :

- $\forall D \in \mathcal{D} \quad \forall f, g \in \mathcal{C}^{\mathcal{D}} \quad \text{si} \quad f \preccurlyeq_D g \quad \text{alors} \quad \forall D' \subset D \quad f \preccurlyeq_{D|D'} g$

Cette condition s'écrit aussi $\preccurlyeq_D \subset \bigcap_{D'\subset D} \preccurlyeq_{D|D'}$.

3.5.4 Compatibilité avec la fusion des supports.

À l'inverse, la *compatibilité avec la fusion des supports* exprime que, si un programme p est meilleur que q *indépendamment* sur des ensembles de données $(D_i)_{i\in I}$, alors p est également meilleur que q sur $\bigcup_{i\in I} D_i$. La fusion faible suppose que $(D_i)_{i\in I}$ est une partition.

Définition 3.37. (Coût compatible avec la fusion (des supports))
Un coût statique $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est *fortement compatible avec la fusion (des supports)* ssi pour tout ensemble I,

- $\forall D \subset \mathcal{D} \ \ \forall (D_i)_{i\in I}$ tq $D_i \subset D \ \ \forall f,g \in \mathcal{C}^D$ si $\forall i \in I \ \ f \preccurlyeq_{D|D_i} g$ alors $f \preccurlyeq_{D|\cup_{i\in I} D_i} g$

Cette condition s'écrit aussi :

- $\forall D \subset \mathcal{D} \ \ \forall (D_i)_{i\in I}$ tq $D_i \subset D \ \ \bigcap_{i\in I} \preccurlyeq_{D|D_i} \subset \preccurlyeq_{D|\cup_{i\in I} D_i}$

Il est *faiblement compatible avec la fusion (des supports)* ssi :

- $\forall D \subset \mathcal{D} \ \ \forall (D_i)_{i\in I}$ tq $D_i \subset D$ si $\forall i,j \in I \ \ i \neq j \ \Rightarrow \ D_i \cap D_j = \varnothing$ alors $\bigcap_{i\in I} \preccurlyeq_{D|D_i} \subset \preccurlyeq_{D|\cup_{i\in I} D_i}$

Il est *fortement* (resp. *faiblement*) *compatible avec la fusion finie (des supports)* lorsque I est fini.

3.5.5 Composabilité.

Beaucoup de transformations sont *locales* ; elles remplacent une portion de programme par une autre. La *composabilité* étudie dans quelles circonstances cette substitution est avantageuse. La remarque à ce sujet que nous avons faite à propos de l'*additivité* (cf. §3.1.2) n'avait de sens que pour une donnée d fixée. Nous formulons un jugement similaire, non plus sur d, mais sur tout un support $D \subset \mathcal{D}$. Pour cela, nous considérerons un langage $L : \mathcal{P} \times \mathcal{D} \to \mathcal{D}$ où résultats et données coïncident. Nous faisons l'hypothèse que les programmes de L sont *composables*.

Définition 3.38. (Programmes composables)
Soit $L : \mathcal{P} \times \mathcal{D} \to \mathcal{D}$ un langage. Les programmes de L sont *composables* ssi :

- $\forall p,q \in \mathcal{P} \ \ \exists (p \circ q) \in \mathcal{P} \ \ \ L\ (p \circ q) = (L\ p) \circ (L\ q)$

Autrement dit, la composition des programmes est exactement la composition des fonctions qu'ils représentent. L'exécution du programme $p \circ q$ sur une donnée d a pour résultat $L\ (p \circ q)\ d = L\ p\ (L\ q\ d)$. Par conséquent, son domaine de définition est :

- $\forall p,q \in \mathcal{P} \ \ \ Dom(p \circ q) = L(q)^{-1}(Dom(p))$

En particulier, $Dom(p \circ q) = Dom(q)$ lorsque $Dom(p) \supset Im(q)$. Nous faisons l'hypothèse supplémentaire que la mesure du coût d'exécution de $L\ (p \circ q)\ d = L\ p\ (L\ q\ d)$ est la somme des coûts d'exécution de $L\ q\ d = d'$ et $L\ p\ d'$.

Définition 3.39. (Mesure de performance composable)
Soit $L : \mathcal{P} \times \mathcal{D} \to \mathcal{D}$ un langage dont les programmes sont composables et μ une mesure de performance sur L. Elle est *composable* ssi :

- $\forall p,q \in \mathcal{P} \ \ \ \mu(p \circ q) = \mu(q) + \mu(p) \circ L(q)$

Autrement dit, $\mu((p \circ q)\ d) = \mu(q\ d) + \mu(p\ (L\ q\ d))$ pour tout $d \in Dom(p \circ q)$. Cette formulation est à peu près identique à celle de Gurr [Gur91] qui considère la composition séquentielle « $p; q$ » de programmes, de dénotation « $[\![p; q]\!] = [\![q]\!][\![p]\!]$ » et de complexité « $cx(p; q) = cx(q) \cdot cx(p)[\![q]\!]$ ».

Lorsque l'on compose deux programmes, on met en rapport le domaine de définition de l'un avec l'image de l'autre. Suivant les inclusions, il en ressort deux variantes de la propriété de composabilité, nombre qu'il faut encore doubler car on peut composer à gauche ou à droite.

Définition 3.40. (Composabilité)
Soit $(\mathcal{C}^D, +, ., \preccurlyeq_D)_{D\subset\mathcal{D}}$ un domaine de coût statique algébrique général.

- Il est *fortement composable à droite* ssi $\forall D \subset \mathcal{D}$
  $\forall f, f' \in \mathcal{C}^D \quad f \preccurlyeq_D f' \;\Rightarrow\; (\forall g \in \mathcal{D}^\mathcal{D} \;\; \forall h \in \mathcal{C}^{\mathrm{Dom}(g)} \quad h + f \circ g \preccurlyeq_{|g^{-1}(D)} h + f' \circ g)$
- Il est *faiblement composable à droite* ssi $\forall D \subset \mathcal{D}$
  $\forall f, f' \in \mathcal{C}^D \quad f \preccurlyeq_D f' \;\Rightarrow\; (\forall g \in \mathcal{D}^\mathcal{D} \;\; \mathrm{Im}(g) \supset D \;\Rightarrow\; \forall h \in \mathcal{C}^{\mathrm{Dom}(g)} \quad h + f \circ g \preccurlyeq_{|g^{-1}(D)} h + f' \circ g)$
- Il est *fortement composable à gauche* ssi $\forall D \subset \mathcal{D}$
  $\forall f, f' \in \mathcal{C}^D \quad f \preccurlyeq_D f' \;\Rightarrow\; (\forall g \in \mathcal{D}^D \;\; \forall h \in \mathcal{C}^\mathcal{D} \quad f + h \circ g \preccurlyeq_{|g^{-1}(\mathrm{Dom}(h))} f' + h \circ g)$
- Il est *faiblement composable à gauche* ssi $\forall D \subset \mathcal{D}$
  $\forall f, f' \in \mathcal{C}^D \quad f \preccurlyeq_D f' \;\Rightarrow\; (\forall g \in \mathcal{D}^D \;\; \forall h \in \mathcal{C}^{g(D)} \quad f + h \circ g \preccurlyeq_D f' + h \circ g)$

Cela se traduit ainsi pour un modèle de coût statique général $(\mathcal{C}^D, \preccurlyeq_D, \mu_D)_{D\subset\mathcal{D}}$ d'un langage $L : \mathcal{P} \times \mathcal{D} \to \mathcal{D}$.

- Il est *fortement composable à droite* ssi $\forall D \subset \mathcal{D}$
  $\forall p, p' \in \mathcal{P}_D \quad p \preccurlyeq_{|D} p' \;\Rightarrow\; (\forall q \in \mathcal{P} \quad p \circ q \preccurlyeq_{|L(q)^{-1}(D)} p' \circ q)$
- Il est *faiblement composable à droite* ssi $\forall D \subset \mathcal{D}$
  $\forall p, p' \in \mathcal{P}_D \quad p \preccurlyeq_{|D} p' \;\Rightarrow\; (\forall q \in \mathcal{P} \quad \mathrm{Im}(q) \supset D \;\Rightarrow\; p \circ q \preccurlyeq_{|L(q)^{-1}(D)} p' \circ q)$
- Il est *fortement composable à gauche* ssi $\forall D \subset \mathcal{D}$
  $\forall p, p' \in \mathcal{P}_D \quad p \preccurlyeq_{|D} p' \;\wedge\; L(p)_{|D} = L(p')_{|D} \;\Rightarrow\; (\forall q \in \mathcal{P} \quad q \circ p \preccurlyeq_{|D\cap\mathrm{Dom}(q\circ p)} q \circ p')$
- Il est *faiblement composable à gauche* ssi $\forall D \subset \mathcal{D}$
  $\forall p, p' \in \mathcal{P}_D \quad p \preccurlyeq_{|D} p' \;\wedge\; L(p)_{|D} = L(p')_{|D} \;\Rightarrow\; (\forall q \in \mathcal{P}_{L(p,D)} \quad q \circ p \preccurlyeq_{|D} q \circ p')$

On pourrait introduire une notion duale de *décomposabilité* en inversant les implications dans les définitions précédentes.

La proposition 3.2, donnée un peu plus loin, montre que la composabilité est intimement reliée à l'additivité et la compatibilité avec la restriction et la fusion des supports.

Sands se pose un problème similaire dans sa formalisation [San93] ; il définit une équivalence de coût sur les programmes et montre que c'est une congruence. Cela correspond à la composabilité à gauche où l'on désire pour tout contexte $C[\cdot]$ que si $p' \preccurlyeq p$ et $q = C[p]$ alors $q' = C[p'] \preccurlyeq q$.

3.5.6 Compatibilité avec les constructions de coûts.

La *compatibilité* exprime qu'un procédé est un morphisme pour les constructions de coût. Elle *étudie* les inclusions en jeu dans les *identités* suivantes, qui ne sont pas nécessairement vérifiées.

- $(\preccurlyeq_{\mathrm{stat}})_{\mathrm{stat}} = \preccurlyeq_{\mathrm{stat}}$
- $(\succcurlyeq)_{\mathrm{stat}} = \succcurlyeq_{\mathrm{stat}}$
- $(\not\preccurlyeq)_{\mathrm{stat}} = \not\preccurlyeq_{\mathrm{stat}}$
- $(\prec)_{\mathrm{stat}} = \prec_{\mathrm{stat}}$
- $(\approx)_{\mathrm{stat}} = \approx_{\mathrm{stat}}$
- Si $\preccurlyeq_1 \subset \preccurlyeq_2$ alors $\preccurlyeq_{1,\mathrm{stat}} \subset \preccurlyeq_{2,\mathrm{stat}}$
- $(\bigcap_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcap_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$
- $(\bigcup_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcup_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$
- $(\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$
- $(\bigotimes_{i\in I}^{\mathrm{lex}} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes_{i\in I}^{\mathrm{lex}} \preccurlyeq_{i,\mathrm{stat}}$

Certaines identités sont peu intéressantes mais simplifient les démonstrations de compatibilité car :

- $\approx \; = \; \preccurlyeq \cap \succcurlyeq$

- $\prec \; = \; \preccurlyeq \cap \not\succcurlyeq$
- $\bigcup_{i\in I} \preccurlyeq_{i,\mathrm{stat}} = \complement(\bigcap_{i\in I} \not\succcurlyeq_{i,\mathrm{stat}})$
- $\bigotimes_{i\in I}^{\mathrm{lex}} \preccurlyeq_i = \bigcup_{i\in I}((\bigotimes_{j\in I, j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I, j>i} \approx_{j,\mathrm{triv}})) \cup (\bigotimes_{i\in I} \preccurlyeq_i)$

La compatibilité avec l'inclusion et la compatibilité avec l'intersection sont reliées car

- $(\preccurlyeq_1 \cap \preccurlyeq_2)_{\mathrm{stat}} \subset \preccurlyeq_{1,\mathrm{stat}} \cap \preccurlyeq_{2,\mathrm{stat}}$ ssi $(\preccurlyeq_1 \subset \preccurlyeq_2 \;\Rightarrow\; \preccurlyeq_{1,\mathrm{stat}} \subset \preccurlyeq_{2,\mathrm{stat}})$

Bien qu'il n'y ait pas dépendance systématique, il y a souvent compatibilité simultanée entre intersection et produit. Cela est dû à l'identité suivante (voir aussi la définition 3.7).

- $\forall c, c' \in \mathcal{C} \quad c\ (\bigcap_{i\in I} \preccurlyeq_i)\ c'$ ssi $(c)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_i)\ (c')_{i\in I}$

En cas d'échec, ces correspondances expliquent également pourquoi une identité n'est pas vérifiée. La compatibilité avec le coût lexicographique est rare car elle demande à ce que beaucoup d'autres propriétés de compatibilité soient satisfaites.

3.5.7 Corrélations entre propriétés.

Il nous faut examiner pour chacun des coûts définis à la section §3.4 lesquelles des qualités précédentes sont vérifiées. Cette tâche est simplifiée si l'on dispose de relations entre qualités. C'est pourquoi nous étudions dans cette section comment elles sont corrélées.

Proposition 3.2. (Liens entre les qualités des relations de coût)
Soit $(\mathcal{C}, +, ., \preccurlyeq)$ un domaine de coût algébrique.

(1) si $\preccurlyeq$ additif et normé alors $\preccurlyeq$ est antisymétrique ; si de plus $\preccurlyeq$ est réflexif (il suffit pour cela que $0 \preccurlyeq 0$ puisque $\preccurlyeq$ est additive) alors $\preccurlyeq$ est une relation d'ordre, et l'équivalence $\approx$ est l'égalité sur $\mathcal{C}$

Soient $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ une relation de coût statique déduite du préordre de coût dynamique $\preccurlyeq$ et D un support.

(2) $\preccurlyeq_D$ réflexif $\Leftarrow$ $\preccurlyeq_D$ stable et réflexif

(3) $\preccurlyeq_D$ discriminant $\Leftarrow$ $\preccurlyeq_D$ ordre stable

(4) $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ compatible avec la fusion faible $\Leftarrow$ $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ compatible avec la fusion forte

(5) $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ compatible avec la fusion forte $\Leftarrow$ $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ compatible avec la fusion faible et la restriction

(6) $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ fortement composable à droite (resp. à gauche) $\Leftrightarrow$ $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ faiblement composable à droite (resp. à gauche) et compatible avec la restriction

(7) $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ faiblement composable à droite $\Leftarrow$ $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ additif et compatible avec la fusion forte.

(8) $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ additif $\Leftarrow$ $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ faiblement composable à droite

(9) $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ faiblement composable à gauche $\Leftrightarrow$ $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ additif

Notons que la composabilité à gauche est *équivalente* à une conjonction de propriétés définies par ailleurs. C'est également « presque » le cas pour la composabilité à droite.

3.6 Propriétés des relations de coûts.

Cette section regroupe les propriétés vérifiées par les diverses relations de coût. Nous donnons quelques identités générales sur les combinaisons de coût (§3.6.1) et indiquons quelles propriétés sont préservées par combinaison (§3.6.2), et quelles propriétés possèdent les exemples de coûts statiques présentés à la section 3.4

(§3.6.3, 3.6.4, 3.6.5). Pour les coût déduits de coûts généraux, nous examinons également si $\preccurlyeq_{\mathrm{stat},D}$ a une forme particulière lorsque le support D est fini. Les tableaux qui regroupent ces propriétés sont placés en fin de chapitre. Comme pour les sections précédentes, les démonstrations, nombreuses et élémentaires, sont repoussées en annexe (cf. §D.3).

3.6.1 Identités des combinaisons de coûts.

Le tableau 3.1 donne des identités générales sur les combinaisons de relations de coût. Il ne fait aucune hypothèse sur la nature des relations $\preccurlyeq$.

Ce tableau ne comportent pas de légende. Nous indiquons par l'exemple comment le lire :

- Ainsi, lorsque $\preccurlyeq \; = \bigcap_{i \in I} \preccurlyeq_i$ (première colonne), alors l'équivalence $\approx$ associée (c'est-à-dire $\approx \; = \; \preccurlyeq \cap \succcurlyeq$) vérifie $\approx \; = \bigcap_{i \in I} \approx_i$ (cinquième colonne) où $\approx_i$ est la relation $\preccurlyeq_i \cap \succcurlyeq_i$.
- De même, la relation stricte $\prec \; = \; \preccurlyeq \cap \not\succcurlyeq$ vérifie (quatrième colonne) $\prec \; \supset \bigcap_{i \in I} \prec_i$.
- Les propriétés de l'inclusion sont lues explicitement : si $\preccurlyeq_1 \subset \preccurlyeq_2$ alors $\approx_1 \subset \approx_2$.
- Une case vide signifie qu'il n'y a pas d'identité particulière concernant la combinaison de coût envisagée.
- Un ensemble vide désigne la relation vide ; par exemple, l'équivalence associée à une relation $\prec$ est $\prec \cap \succ \; = \; \preccurlyeq \cap \not\succcurlyeq \cap \succcurlyeq \cap \not\preccurlyeq \; = \varnothing$.

Seul ce tableau-ci représente des équations entre relations de coûts. Les tableaux des sections suivantes expriment si une propriété est vérifiée ou non.

En fait, ces identités sont la simple expression d'opérations élémentaires sur les ensembles ou, de manière équivalente, de manipulation de formules logiques – nous n'en faisons pas la démonstration. Elles ont pour but principal de simplifier la démonstration des propriétés qui suivent.

3.6.2 Propriétés des combinaisons de coûts.

Dans le mesure où certains coûts statiques sont donnés en terme de combinaisons, nous indiquons tout d'abord, à l'aide du tableau 3.2, quelles sont les propriétés qu'elles préservent. Il sera plus simple ensuite d'en déduire les propriétés préservées par les procédés de construction de coûts statiques. Les propriétés suivantes, qui portent simultanément sur $\preccurlyeq$ et $\prec$, ne figurent pas dans le tableau.

Proposition 3.3. (Propriétés annexes des combinaisons de coût)
Soit $(\mathcal{C}, +, ., \preccurlyeq)$ un domaine de coût algébrique.

- $\prec \cdot \preccurlyeq \, , \; \preccurlyeq \cdot \prec \; \subset \; \prec$
- $\preccurlyeq + \prec \; = \; \prec + \preccurlyeq \; \subset \; \prec$ si $\preccurlyeq$ est additif

Soit $\preccurlyeq_{\mathrm{stat}}$ un coût statique déduit d'un coût dynamique $\preccurlyeq$.

- $\prec_{\mathrm{abs}} \subset \prec_{\mathrm{stat}}$ si $\preccurlyeq_{\mathrm{stat}}$ est stable et discriminant

3.6.3 Propriétés préservées par les coûts statiques déduits de coûts dynamiques.

Le tableau 3.3 indique quelles propriétés sont préservées par la construction de coûts statiques à partir de coûts dynamiques. On peut y lire par exemple que $\preccurlyeq_{\mathrm{moy}}$ et $\preccurlyeq_{\mathrm{max}}$ sont stables. Du tableau 3.2, on peut conclure que $\preccurlyeq_{\text{moy-max}}$ est également stable.

3.6.4 Propriétés préservées par les coûts statiques déduits de coûts statiques généraux.

Le tableau 3.4 indique quelles propriétés sont préservées par la construction de coûts statiques à partir de coûts statiques généraux. On peut remarquer que la restriction $\preccurlyeq_{|\Delta}$ qui apparaît dans la définition de plusieurs coûts statiques fait perdre le caractère discriminant (cf. §3.5.2). Seuls les coûts $\preccurlyeq_{\text{fini}}$, $\preccurlyeq_{\text{pfini}}$, et $\preccurlyeq_{\text{asymglob}}$ le conservent malgré tout. De même la composabilité à droite n'est pas préservée par restriction. Pour les coûts définis à partir de coûts généraux finis, cela est dû fait que si $(f(d))_{d\in Dom(f)}$ est fini, la famille $((f\circ g)(d))_{d\in Dom(f\circ g)}$, dont les éléments sont pourtant inclus dans $(f(d))_{d\in Dom(f)}$, n'est pas nécessairement finie.

3.6.5 Propriétés préservées par les coûts statiques déduits de coûts statiques.

Le tableau 3.5 indique quelles propriétés sont préservées par la construction de coûts statiques à partir de coûts statiques. Notons que pondérer par une distribution affecte bien évidemment la hiérarchie de comparaison des programmes, mais ne modifie pas la classification relative des échelles de coût.

3.6.6 Graduation des coûts.

L'énumération de ces coûts constitue une *échelle de comparaison*, selon laquelle *graduer* la finesse d'une transformation. De plus, connaître les propriétés d'inclusion permet de composer des transformations de gain différent :

- si $p_0 \succcurlyeq_1 p_1 \succcurlyeq_2 p_2$ et $\preccurlyeq_1 \subset \preccurlyeq_2$ alors $p_0 \succcurlyeq_2 p_2$

Les propositions suivantes indiquent comment les coûts définis précédemment sont corrélés. Pour toutes ces propositions, lorsque l'expression d'un coût repose sur des hypothèses (additivité, finitude du support, topologie, taille, etc.), nous les supposons vérifiées au moment de juger chaque inclusion.

Coût statique déduit d'un coût statique général.

(1) $\preccurlyeq_{\text{fini}} \subset \preccurlyeq_{\text{abs-pp}} \subset^1 \preccurlyeq_{\text{pp}} \subset \preccurlyeq_{\text{asymloc}}$

(2) $\preccurlyeq_{\text{fini}} \subset^{1,2} \preccurlyeq_{\text{pfini}} \subset \preccurlyeq_{\text{asymglob}} \subset^3 \preccurlyeq_{\text{asymloc}}$

Si $(\preccurlyeq_D)_{D\in\mathcal{D}}$ est (1.) stable / (2.) compatible avec la fusion faible / (3.) compatible avec la restriction.

Distribution.

(3) $(\preccurlyeq^{\nu_1})^{\nu_2} = \preccurlyeq^{\nu_1.\nu_2}$

(4) $\preccurlyeq_{\text{abs}} \subset^1 \preccurlyeq^{\nu}_{\text{abs}}$ et $\preccurlyeq_{\text{abs-pp}} \subset^1 \preccurlyeq^{\nu}_{\text{abs-pp}}$

(5) $(\preccurlyeq_{D|D'})^{\nu} = (\preccurlyeq^{\nu}_{D})_{|D'}$

Notons (1.) que la réciproque est vraie si $\nu > 0$.

Coût limite.

(6) $\preccurlyeq_D \subset \preccurlyeq_{\text{lim},D}$

Coût maximum.

(7) $\preccurlyeq_{\max} \subset \preccurlyeq_{\text{max-asymglob}}$

(8) $\preccurlyeq_{\text{abs}} \subset^1 \preccurlyeq_{\text{max-asymloc}}$, $\preccurlyeq_{\text{max-asymglob}}$

Sous l'hypothèse (1.) que $(\preccurlyeq_D)_{D\in\mathcal{D}}$ est totale.

Coûts majorés. Soient $1 < \alpha_1, \alpha_2$.

(9) $(\preccurlyeq_{\mathrm{maj}(\alpha_1)})_{\mathrm{maj}(\alpha_2)} = \preccurlyeq_{\mathrm{maj}(\alpha_1\alpha_2)}$

(10) $(\preccurlyeq_{\mathrm{majl}(\alpha_1)})_{\mathrm{maj}(\alpha_2)} = (\preccurlyeq_{\mathrm{maj}(\alpha_1)})_{\mathrm{majl}(\alpha_2)} = (\preccurlyeq_{\mathrm{majl}(\alpha_1)})_{\mathrm{majl}(\alpha_2)} = \preccurlyeq_{\mathrm{majl}(\alpha_1\alpha_2)}$

(11) $\preccurlyeq_{\mathrm{maj}(\alpha_1)} \cdot \preccurlyeq_{\mathrm{maj}(\alpha_2)} \subset^1 \preccurlyeq_{\mathrm{maj}(\alpha_1\alpha_2)}$

(12) $\preccurlyeq_{\mathrm{maj}(\alpha_1)} \cdot \preccurlyeq_{\mathrm{majl}(\alpha_2)}\ ,\ \preccurlyeq_{\mathrm{majl}(\alpha_1)} \cdot \preccurlyeq_{\mathrm{maj}(\alpha_2)}\ ,\ \preccurlyeq_{\mathrm{majl}(\alpha_1)} \cdot \preccurlyeq_{\mathrm{majl}(\alpha_2)} \subset^1 \preccurlyeq_{\mathrm{majl}(\alpha_1\alpha_2)}$

(13) si $1 < \alpha_1 < \alpha_2$ alors $\preccurlyeq_{\text{négl}} \subset \preccurlyeq_{\mathrm{maj}(\alpha_2)} \subset \preccurlyeq_{\mathrm{majl}(\alpha_2)} \subset \preccurlyeq_{\mathrm{maj}(\alpha_1)} \subset \preccurlyeq_{\mathrm{majl}(\alpha_1)}$

(14) $\preccurlyeq_{\mathrm{maj}(\alpha_1)} \cap \preccurlyeq_{\mathrm{maj}(\alpha_2)} = \preccurlyeq_{\mathrm{maj}(\max(\alpha_1,\alpha_2))}$

(15) $\preccurlyeq_{\mathrm{majl}(\alpha_1)} \cap \preccurlyeq_{\mathrm{majl}(\alpha_2)} = \preccurlyeq_{\mathrm{majl}(\max(\alpha_1,\alpha_2))}$

Sous l'hypothèse (1.) que $\preccurlyeq_D$ est homothétique.

Support fini.

(16) $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\mathrm{moy}} = \preccurlyeq_{\text{moy-lim}}$

(17) $\preccurlyeq_{\mathrm{abs}} \subset^1 \preccurlyeq_{\mathrm{max}} = \preccurlyeq_{\text{max-lim}}$

(18) $\preccurlyeq_{\mathrm{pp}} = \preccurlyeq_{\mathrm{asymloc}} = \approx_{\mathrm{triv}}$

(19) $\preccurlyeq_{\mathrm{fini}} = \preccurlyeq_{\mathrm{pfini}} = \preccurlyeq_{\mathrm{asymglob}} = \preccurlyeq_D$

Sous l'hypothèse (1.) que $(\preccurlyeq_D)_{D\in\mathcal{D}}$ est totale.

Coûts d'usage pratique. En particulier, les coûts d'usage pratique vérifient :

$$\preccurlyeq_{\mathrm{abs}} \subset \begin{cases} {}^1\preccurlyeq_{\text{max-asymloc}} \\ {}^1\preccurlyeq_{\text{max-asymglob}} \\ \preccurlyeq_{\text{moy-fini}} \subset \begin{cases} \preccurlyeq_{\text{abs-pp}} \subset \preccurlyeq_{\text{moy-pp}} \subset \preccurlyeq_{\text{moy-asymloc}} \\ \preccurlyeq_{\text{moy-pfini}} \subset \begin{cases} \preccurlyeq_{\text{moy-asymglob}} \\ \preccurlyeq_{\text{moy-lim}} \end{cases} \end{cases} \end{cases}$$

sous l'hypothèse (1.) que $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est totale. De plus, toutes ces inclusions sont strictes en général : il n'y a pas d'inclusions en dehors de celles indiquées.

Dans le cas D fini, ces coûts se réduisent à :

$$\preccurlyeq_{\mathrm{abs}} \subset \left\{ \begin{array}{l} {}^1\preccurlyeq_{\mathrm{max}} = \preccurlyeq_{\text{max-asymglob}} \\ \preccurlyeq_{\mathrm{moy}} = \preccurlyeq_{\text{moy-fini}} = \preccurlyeq_{\text{moy-pfini}} = \preccurlyeq_{\text{moy-lim}} = \preccurlyeq_{\text{moy-asymglob}} \end{array} \right\} \subset \preccurlyeq_{\mathrm{asymloc}} = \preccurlyeq_{\mathrm{pp}} = \approx_{\mathrm{triv}}$$

sous l'hypothèse (1.) que $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est totale.

Ces inclusions chaînées fournissent une graduation pour des transformations. Par exemple, une transformation T telle que pour tout programme p on ait $T(p) \preccurlyeq_{\text{abs-fini}} p$ est meilleure qu'une transformation qui vérifie $T'(p) \preccurlyeq_{\text{moy-fini}} p$, mais moins bonne qu'une transformation vérifiant $T''(p) \preccurlyeq_{\mathrm{abs}} p$. Néanmoins, il n'existe pas nécessairement de relations entre les programmes $T(p)$, $T'(p)$, et $T''(p)$.

Conclusion du chapitre.

Nous avons proposé une définition formelle de modèle de performance capable de traiter des aspects comme le temps d'exécution ou l'espace mémoire consommé (§3.1, 3.2, 3.3), et défini une mesure de performance générique sur les classes de modèles d'exécution définies au chapitre précédent à l'aide de la sémantique naturelle et des schémas de programme (§3.2.5, 3.2.6).

Nous avons construit (§3.4) plusieurs modèles de coût statique (coût absolu, moyen, maximal), et défini des moyens de déduire un coût statique à partir d'un ou plusieurs autres (distribution, coût presque partout, fini, presque fini, limite, asymptotique, majoré).

Nous avons étudié quelles sont les qualités souhaitables d'un modèle de coût statique (§3.5), et en particulier les propriétés de stabilité (déf. 3.34) et de discrimination (déf. 3.35). Nous avons indiqué comment ces qualités sont reliées entre elles, et notamment les relations entre composabilité et additivité (prop. 3.2).

Enfin, nous avons minutieusement et systématiquement recensé lesquelles parmi ces qualités étaient apportées ou préservées par les procédés de construction de coût statique (§3.6). Nous avons également établi les propriétés d'inclusion de ces modèles de coûts afin de fournir des échelles de comparaison (§3.6.6).

Cette notion de modèle de performance nous permet de formaliser au chapitre suivant le concept d'évaluation partielle.

$\preccurlyeq$	$\succcurlyeq$	$\not\preccurlyeq$	$\prec$	$\approx$
$\succcurlyeq$	$\preccurlyeq$	$\not\succcurlyeq$	$\succ$	$\approx$
$\not\preccurlyeq$	$\not\succcurlyeq$	$\preccurlyeq$	$\succ$	$\subset\not\approx$
$\prec$	$\succ$	$\supset\succ$	$\prec$	$\varnothing$
$\approx$	$\approx$		$\varnothing$	$\approx$
$\preccurlyeq_1\subset\preccurlyeq_2$	$\succcurlyeq_1\subset\succcurlyeq_2$	$\not\preccurlyeq_1\supset\not\preccurlyeq_2$		$\approx_1\subset\approx_2$
$\bigcap_{i\in I}\preccurlyeq_i$	$\bigcap_{i\in I}\succcurlyeq_i$	$\supset\bigcap_{i\in I}\not\preccurlyeq_i$	$\supset\bigcap_{i\in I}\prec_i$	$\bigcap_{i\in I}\approx_i$
$\bigcup_{i\in I}\preccurlyeq_i$	$\bigcup_{i\in I}\succcurlyeq_i$	$\subset\bigcup_{i\in I}\not\preccurlyeq_i$	$\subset\bigcup_{i\in I}\prec_i$	$\supset\bigcup_{i\in I}\approx_i$
$\bigotimes_{i\in I}\preccurlyeq_i$	$\bigotimes_{i\in I}\succcurlyeq_i$	$\supset\bigotimes_{i\in I}\not\preccurlyeq_i$	$\supset\bigotimes_{i\in I}\prec_i$	$\bigotimes_{i\in I}\approx_i$
$\overset{\text{lex}}{\bigotimes_{i\in I}}\preccurlyeq_i$	$\overset{\text{lex}}{\bigotimes_{i\in I}}\succcurlyeq_i$			$\bigotimes_{i\in I}\approx_i$

$\preccurlyeq^{(\prime)}$	$\preccurlyeq\cdot\preccurlyeq'$	$\preccurlyeq+\preccurlyeq'$	$\alpha.\preccurlyeq$
$\preccurlyeq_1\subset\preccurlyeq_2$	$(\preccurlyeq_1\cdot\preccurlyeq_1')\subset(\preccurlyeq_2\cdot\preccurlyeq_2')$	$(\preccurlyeq_1+\preccurlyeq_1')\subset(\preccurlyeq_2+\preccurlyeq_2')$	$\alpha.\preccurlyeq_1\subset\alpha.\preccurlyeq_2$
$\bigcap_{i\in I}\preccurlyeq_i$	$\subset\bigcap_{i\in I}(\preccurlyeq_i\cdot\preccurlyeq_i')$	$\subset\bigcap_{i\in I}(\preccurlyeq_i+\preccurlyeq_i')$	$\subset\bigcap_{i\in I}\alpha.\preccurlyeq_i$
$\bigcup_{i\in I}\preccurlyeq_i$	$\supset\bigcup_{i\in I}(\preccurlyeq_i\cdot\preccurlyeq_i')$	$\bigcup_{i\in I}(\preccurlyeq_i+\preccurlyeq')$	$\bigcup_{i\in I}\alpha.\preccurlyeq_i$
$\bigotimes_{i\in I}\preccurlyeq_i$	$\bigotimes_{i\in I}(\preccurlyeq_i\cdot\preccurlyeq_i')$	$\bigotimes_{i\in I}(\preccurlyeq_i+\preccurlyeq_i')$	$\bigotimes_{i\in I}\alpha.\preccurlyeq_i$
$\overset{\text{lex}}{\bigotimes_{i\in I}}\preccurlyeq_i$			$\subset\overset{\text{lex}}{\bigotimes_{i\in I}}\alpha.\preccurlyeq_i$

$\preccurlyeq^j$	$\bigcap_{j\in J}\preccurlyeq^j$	$\bigcup_{j\in J}\preccurlyeq^j$	$\bigotimes_{j\in J}\preccurlyeq^j$	$\overset{\text{lex}}{\bigotimes_{j\in J}}\preccurlyeq^j$
$\preccurlyeq_1\subset\preccurlyeq_2$	$\bigcap_{j\in J}\preccurlyeq_1^j\subset\bigcap_{j\in J}\preccurlyeq_2^j$	$\bigcup_{j\in J}\preccurlyeq_1^j\subset\bigcup_{j\in J}\preccurlyeq_2^j$	$\bigotimes_{j\in J}\preccurlyeq_1^j\subset\bigotimes_{j\in J}\preccurlyeq_2^j$	
$\bigcap_{i\in I}\preccurlyeq_i$	$\bigcap_{i\in I}(\bigcap_{j\in J}\preccurlyeq_i^j)$	$\subset\bigcap_{i\in I}(\bigcup_{j\in J}\preccurlyeq_i^j)$	$\bigcap_{i\in I}(\bigotimes_{j\in J}\preccurlyeq_i^j)$	$\supset\bigcap_{i\in I}(\overset{\text{lex}}{\bigotimes_{j\in J}}\preccurlyeq_i^j)$
$\bigcup_{i\in I}\preccurlyeq_i$	$\supset\bigcup_{i\in I}(\bigcap_{j\in J}\preccurlyeq_i^j)$	$\bigcup_{i\in I}(\bigcup_{j\in J}\preccurlyeq_i^j)$	$\supset\bigcup_{i\in I}(\bigotimes_{j\in J}\preccurlyeq_i^j)$	
$\bigotimes_{i\in I}\preccurlyeq_i$	$\bigotimes_{i\in I}(\bigcap_{j\in J}\preccurlyeq_i^j)$	$\subset\bigotimes_{i\in I}(\bigcup_{j\in J}\preccurlyeq_i^j)$	$\bigotimes_{i\in I}\bigotimes_{j\in J}\preccurlyeq_i^j$	
$\overset{\text{lex}}{\bigotimes_{i\in I}}\preccurlyeq_i$	$\subset\overset{\text{lex}}{\bigotimes_{i\in I}}(\bigcap_{j\in J}\preccurlyeq_i^j)$			

Tableau 3.1 : Identités des combinaisons de coûts

Propriété de $\preccurlyeq_{(i)}$	$\succcurlyeq$	$\not\preccurlyeq$	$\prec$	$\approx$	$\preccurlyeq_1 \subset \preccurlyeq_2$	$\bigcap_{i\in I} \preccurlyeq_i$	$\bigcup_{i\in I} \preccurlyeq_i$	$\bigotimes_{i\in I} \preccurlyeq_i$	$\bigotimes_{i\in I}^{\text{lex}} \preccurlyeq_i$
réflexivité	$\checkmark$			$\checkmark$	$\Rightarrow$	$\checkmark$	$\checkmark^*$	$\checkmark$	$\checkmark$
antisymétrie	$\checkmark$		$\checkmark^4$	$\checkmark$	$\Leftarrow$	$\checkmark^*$		$\checkmark$	$\checkmark$
transitivité	$\checkmark$		$\checkmark$	$\checkmark$		$\checkmark$	$\checkmark^5$	$\checkmark$	$\checkmark$
totalité	$\checkmark$				$\Rightarrow$		$\checkmark^*$		
additivité	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$		$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
homothéticité	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$		$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
stabilité	$\checkmark$			$\checkmark$	$\Rightarrow$	$\checkmark$	$\checkmark^*$	$\checkmark$	$\checkmark$
discrimination	$\checkmark$		$\checkmark$	$\checkmark$	$\Leftarrow$	$\checkmark^*$	$\checkmark$	$\checkmark$	
restriction	$\checkmark$			$\checkmark$		$\checkmark$	$\checkmark$	$\checkmark$	
fusion[1]	$\checkmark$	$\checkmark^3$	$\checkmark^3$	$\checkmark$		$\checkmark$		$\checkmark$	
composabilité[1,2]	$\checkmark$			$\checkmark$		$\checkmark$	$\checkmark$	$\checkmark$	

$\checkmark$: si la propriété est vérifiée par $\preccurlyeq$, elle l'est aussi par $\succcurlyeq$, $\not\preccurlyeq$, $\prec$, $\approx$
$\checkmark$: si la propriété est vérifiée par tous les $\preccurlyeq_i$, elle l'est aussi par $\cdots_{i\in I} \preccurlyeq_i$
$\checkmark^*$: si la propriété est vérifiée par un des $\preccurlyeq_i$, elle l'est aussi par $\cdots_{i\in I} \preccurlyeq_i$
$\Rightarrow$: si la propriété est vérifiée par $\preccurlyeq_1$, elle l'est aussi par $\preccurlyeq_2$
$\Leftarrow$: si la propriété est vérifiée par $\preccurlyeq_2$, elle l'est aussi par $\preccurlyeq_1$
1. : forte (resp. faible)
2. : à gauche (resp. à droite)
3. : si $\preccurlyeq$ est compatible avec la restriction des supports
4. : triviale (l'équivalence associée est vide)
5. : si $\forall i, j \in I \quad \exists k \in I \quad \preccurlyeq_i \cup \preccurlyeq_j \subset \preccurlyeq_k$; c'est vrai en particulier lorsque $(\preccurlyeq_i)_{i\in I}$ est une suite croissante.

Tableau 3.2 : Propriétés des combinaisons de coûts

Propriété de $\preccurlyeq$ $\quad$ $\mathrm{stat} =$	abs	moy^1	max
support nécessairement fini		$\checkmark$	
réflexivité	$\checkmark$	$\checkmark$	$\checkmark^{2,3}$
antisymétrie	$\checkmark$		
transitivité	$\checkmark$	$\checkmark$	$\checkmark$
totalité		$\checkmark$	$\checkmark^{3}$
additivité	$\checkmark$	$\checkmark$	
homothéticité	$\checkmark$	$\checkmark$	$\checkmark$
stabilité	$\checkmark$	$\checkmark^{8}$	$\checkmark^{2,3}$
discrimination	$\checkmark$	$\checkmark^{8}$	
restriction	$\checkmark$		
fusion	forte	faible^{4}	forte^{2,5}
composabilité	forte^{6}	faible^{7,8}	
$(\succcurlyeq)_{\mathrm{stat}} = \succcurlyeq_{\mathrm{stat}}$	$\checkmark$	$\checkmark$	
$(\not\preccurlyeq)_{\mathrm{stat}} = \not\preccurlyeq_{\mathrm{stat}}$	$\subset$	$\checkmark$	
$(\prec)_{\mathrm{stat}} = \prec_{\mathrm{stat}}$	$\subset$	$\checkmark$	
$(\approx)_{\mathrm{stat}} = \approx_{\mathrm{stat}}$	$\checkmark$	$\checkmark$	
$\preccurlyeq_1 \subset \preccurlyeq_2 \ \Rightarrow\ \preccurlyeq_{1,\mathrm{stat}} \subset \preccurlyeq_{2,\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark$
$(\bigcap_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcap_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\subset$
$(\bigcup_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcup_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\supset$	$\checkmark$	
$(\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark$
$(\bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$		$\checkmark$	

1. D est par hypothèse fini
2. si $\preccurlyeq$ est total
3. si D fini
4. nécessairement finie car D est fini
5. fusion finie
6. à gauche et à droite
7. à gauche
8. si $\preccurlyeq$ est additif

Tableau 3.3 : Propriétés préservées par les coûts statiques $\preccurlyeq_{\mathrm{stat},D}$ déduits d'un coût dynamique $\preccurlyeq$

Propriété de $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ stat =	$\lvert D'$	pp	fini	pfini	asymloc	asymglob
supports finis suffisants			$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
réflexivité	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
antisymétrie			$\checkmark^{2}$	$\checkmark$		$\checkmark$
transitivité	$\checkmark$	$\checkmark^{1}$	$\checkmark^{2}$	$\checkmark$	$\checkmark$	$\checkmark$
totalité						
additivité	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
homothéticité	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
stabilité	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
discrimination			$\checkmark$	$\checkmark$		$\checkmark$
restriction	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
fusion forte[3]	$\checkmark$	$\checkmark^{5}$	$\checkmark^{5}$	$\checkmark^{5}$	$\checkmark^{5}$	$\checkmark^{5}$
composabilité[3,4]	$\checkmark^{6}$	$\checkmark^{6}$	$\checkmark^{6}$	$\checkmark^{6}$	$\checkmark^{6}$	$\checkmark^{6}$
$\preccurlyeq_{\mathrm{stat},D}$ si D est fini		$\approx_{\mathrm{triv}}$	${\preccurlyeq_D}^{2}$	$\preccurlyeq_D$	$\approx_{\mathrm{triv}}$	$\preccurlyeq_D$
$(\preccurlyeq_{\mathrm{stat}})_{\mathrm{stat}} = \preccurlyeq_{\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\approx_{\mathrm{triv}}$	$\checkmark$
$(\succcurlyeq)_{\mathrm{stat}} = \succcurlyeq_{\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
$(\not\succcurlyeq)_{\mathrm{stat}} = \not\succcurlyeq_{\mathrm{stat}}$	$\checkmark$	$\supset$	$\subset$	$\subset$	$\subset$	$\subset$
$(\prec)_{\mathrm{stat}} = \prec_{\mathrm{stat}}$	$\checkmark$	$\supset$	$\subset$	$\subset$	$\subset$	$\subset$
$(\approx)_{\mathrm{stat}} = \approx_{\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
$\preccurlyeq_1 \subset \preccurlyeq_2 \ \Rightarrow\ \preccurlyeq_{1,\mathrm{stat}} \subset \preccurlyeq_{2,\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
$(\bigcap_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcap_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\checkmark$	$\subset^{9}$	$\subset^{10}$	$\subset^{8}$	$\subset^{8}$	$\subset^{8}$
$(\bigcup_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcup_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\checkmark$	$\subset^{9}$	$\supset^{2,7}$	$\supset^{7}$	$\supset^{7}$	$\supset^{7}$
$(\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\checkmark$	$\subset^{9}$	$\subset^{10}$	$\subset^{8}$	$\subset^{8}$	$\subset^{8}$
$(\bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\checkmark$	$\supset^{1,7}$				

1. si $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est compatible avec la restriction du support
2. si $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est stable et compatible avec la fusion faible finie des supports
3. forte (resp. faible)
4. à gauche (resp. à droite)
5. fusion finie
6. à gauche uniquement
7. si I est fini
8. on a égalité si (7.)
9. on a égalité si (7.) et (1.)
10. on a égalité si (7.) et (2.)

Tableau 3.4 : Propriétés préservées par les coûts statiques $\preccurlyeq_{\mathrm{stat},D}$ déduits de coûts statiques généraux $(\preccurlyeq_D)_{D\subset\mathcal{D}}$

Propriété de $\preccurlyeq_D$ $\quad$ $\mathrm{stat} =$	lim^1	ν	$\mathrm{maj}(\alpha)^3$	$\mathrm{majl}(\alpha)^3$	maj	$\mathrm{négl}$
supports finis suffisants						
réflexivité	$\checkmark$	$\checkmark$				
antisymétrie		$\checkmark^{7}$	$\checkmark^{15}$	$\checkmark^{15}$	$\checkmark^{8,15}$	$\checkmark^{15}$
transitivité	$\checkmark^{2}$	$\checkmark$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$
totalité		$\checkmark$				
additivité	$\checkmark$	$\checkmark$				
homothéticité	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
stabilité	$\checkmark$	$\checkmark$				
discrimination	$\checkmark^{2,4,14}$	$\checkmark^{7}$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
restriction	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
fusion[10]	$\checkmark^{2,4,5,9}$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark^{9}$	$\checkmark$
composabilité[10,11]	$\checkmark^{2,12}$	$\checkmark$				
$(\preccurlyeq_{\mathrm{stat}})_{\mathrm{stat}} = \preccurlyeq_{\mathrm{stat}}$	$\checkmark$		$\subset$	$\subset$	$\checkmark$	$\checkmark$
$(\succcurlyeq)_{\mathrm{stat}} = \succcurlyeq_{\mathrm{stat}}$	$\checkmark$	$\checkmark$				
$(\not\preccurlyeq)_{\mathrm{stat}} = \not\preccurlyeq_{\mathrm{stat}}$		$\checkmark$	$\subset$	$\subset$		$\subset$
$(\prec)_{\mathrm{stat}} = \prec_{\mathrm{stat}}$		$\checkmark$	$\subset^{8}$	$\subset^{8}$	$\subset$	$\subset^{8}$
$(\approx)_{\mathrm{stat}} = \approx_{\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark^{15}$	$\checkmark^{15}$	$\checkmark^{15}$	$\checkmark^{15}$
$\preccurlyeq_1 \subset \preccurlyeq_2 \;\Rightarrow\; \preccurlyeq_{1,\mathrm{stat}} \subset \preccurlyeq_{2,\mathrm{stat}}$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
$(\bigcap_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcap_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\subset^{6}$	$\checkmark$	$\checkmark$	$\checkmark$	$\subset^{13}$	$\checkmark$
$(\bigcup_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} = \bigcup_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\supset^{13}$	$\checkmark$	$\supset$	$\supset$	$\supset$	$\supset$
$(\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{stat}}$	$\subset^{13}$	$\checkmark$	$\checkmark$	$\checkmark$	$\subset^{13}$	$\checkmark$
$(\bigotimes_{i\in I}^{\mathrm{lex}} \preccurlyeq_i)_{\mathrm{stat}} \simeq \bigotimes_{i\in I}^{\mathrm{lex}} \preccurlyeq_{i,\mathrm{stat}}$		$\checkmark$				

1. $\preccurlyeq_D$ est par hypothèse normé
2. si $\preccurlyeq_D$ est additif
3. $\alpha > 1$ par hypothèse
4. si $\preccurlyeq_D$ est stable
5. fusion faible ; forte si $\mathcal{C}^D$ est treillissé
6. il y a égalité lorsque $\mathcal{C}^D$ est treillissé, $\preccurlyeq_D$ est stable, et I est fini
7. si $\nu > 0$
8. si $\preccurlyeq_D$ est homothétique
9. fusion finie
10. forte (resp. faible)
11. à gauche (resp. à droite)
12. à gauche uniquement
13. il y a égalité si I est fini
14. si $\preccurlyeq_D$ est réflexif
15. triviale car $\approx_{\mathrm{stat}} = \{(0,0)\}$ si $\preccurlyeq_D$ est additif et homothétique

Tableau 3.5 : Propriétés des coûts statiques $\preccurlyeq_{\mathrm{stat},D}$ déduits de coûts statiques $\preccurlyeq_D$

Chapitre 4

Optimisation et Optimalité

Le terme d'*optimisation* a quelque peu perdu son sens premier de *recherche d'optimum*, pour ne plus signifier bien souvent qu'*amélioration*. Cette dérive du superlatif au comparatif est somme toute assez naturelle dans la mesure où la complexité des problèmes interdit souvent tout espoir d'optimum : améliorer, c'est déjà beaucoup.

Nous distinguons dans ce chapitre l'*évaluation partielle*, qui *optimise* (c'est-à-dire améliore) un programme p donné, et l'*évaluation partielle optimale*, qui recherche les *meilleures optimisations* de p. Car il nous semble raisonnable de chercher des améliorations *optimales* pour des classes de programmes relativement simples, et de nous contenter de modestes *mieux* pour les classes plus complexes. C'est ce qui justifie l'ingénuité de nos exemples.

Organisation du chapitre.

- §4.1 Nous donnons des définitions formelles concernant l'*équivalence de programmes*.
- §4.2 Nous définissons ensuite plusieurs notions d'*optimalité* et en donnons quelques propriétés.
- §4.3 Grâce à ces deux notions sont formellement définies et étudiées l'*évaluation partielle*, ainsi que divers types d'*évaluations partielles optimales*.
- §4.4 Nous illustrons ces définitions à l'aide des modèles de performance définis au chapitre précédent.
- §4.5 Les cas d'évaluation partielle optimale sont rares. Nous en examinons des *conditions d'existence* et d'*inexistence*. Nous constatons également l'échec ou les limites de plusieurs *méthodes*.
- §4.6 Nous étudions comment *transporter* des évaluations partielles, éventuellement optimales, d'un langage dans un autre. L'objectif est de pouvoir résoudre le problème d'abord dans un langage plus simple, puis d'en transporter des solutions dans un langage plus complexe.
- §4.7 Nous discutons ensuite les liens entre évaluation partielle optimale et *compilation optimale*.
- §4.8 Pour conclure, nous donnons informellement quelques principes qui sont à la base des techniques d'optimisations et évoquons le compromis espace statique/temps dynamique. Nous disons également quel sens donner à l'optimisation d'un *ensemble d'exécution*.

Une évaluation partielle est obtenue par transformation de programme. Le chapitre suivant étudie des conditions de correction pour quelques transformations.

4.1 Programmes et équivalence.

L'évaluation partielle d'un programme p est avant tout un programme *équivalent à* p. Nous rappelons (cf. §1.1) et précisons quelques notations sur l'équivalence de programmes.

Définition 4.1. (Équivalence de programmes)
Soient $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ un langage et $D \subset \mathcal{D}$ un ensemble de données. L'*équivalence stricte*, l'*extension paresseuse de l'équivalence* et l'*équivalence sur un support* sont définies respectivement ainsi :

- $\forall p, q \in \mathcal{P} \quad p \equiv q \quad \text{ssi} \quad L(p) = L(q)$
- $\forall p, q \in \mathcal{P} \quad p \sqsubseteq q \quad \text{ssi} \quad L(p) = L(q)_{|Dom(p)}$
- $\forall p, q \in \mathcal{P}_D \quad p \equiv_D q \quad \text{ssi} \quad L(p)_{|D} = L(q)_{|D}$

L'ensemble des programmes *strictement* (resp. *paresseusement*, resp. *sur* D) *équivalents à* $p \in \mathcal{P}$ est défini par :

- $Equiv_{\text{str}}(p) = \{q \in \mathcal{P} \mid p \equiv q\}$
- $Equiv_{\text{par}}(p) = \{q \in \mathcal{P} \mid p \sqsubseteq q\}$
- $Equiv_D(p) = \{q \in \mathcal{P}_D \mid p \equiv_D q\}$ pour $p \in \mathcal{P}_D$

On a aussi $p \equiv q$ ssi $p \sqsubseteq q$ et $p \sqsupseteq q$.

Voici quelques propriétés élémentaires de cette équivalence de programme, qui permettent notamment de comparer entre eux des ensembles de programmes équivalents.

Proposition 4.1. (Propriétés sur l'équivalence des programmes)
Soient $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ un langage et $D \subset \mathcal{D}$ un ensemble de données.

- $\sqsubseteq$ est un préordre sur $\mathcal{P}$
- $\equiv$ est une relation d'équivalence sur $\mathcal{P}$
- $\equiv_D$ est une relation d'équivalence sur $\mathcal{P}_D$

Soient $p, q \in \mathcal{P}$.

- Si $p \sqsubseteq q$ alors $Equiv_{\text{str}}(q) \subset Equiv_{\text{par}}(q) \subset Equiv_{\text{par}}(p) \subset \mathcal{P}_{|Dom(p)}$
- Si $p \sqsubseteq q$ alors $Equiv_{\text{str}}(p) \sqsubseteq Equiv_{\text{str}}(q) \sqsubseteq Equiv_{\text{par}}(q)$

Soient $D' \subset D \subset \mathcal{D}$.

- $\forall p \in \mathcal{P}_D \quad Equiv_{\text{str}}(p) \subset Equiv_{\text{par}}(p) \subset Equiv_D(p) \subset Equiv_{D'}(p)$
- $\forall p \in \mathcal{P}_D \quad Equiv_{\text{str}}(p) \equiv_D Equiv_{\text{par}}(p) \equiv_D Equiv_D(p)$

Si $L : \mathcal{P} \to (\mathcal{D} \to \mathcal{R})$ est une injection, alors $\forall p \in \mathcal{P} \quad Equiv_{\text{str}}(p) = \{p\}$.

Équivalence et composabilité (cf. §3.5.5) sont reliées ainsi.

Proposition 4.2. (Liens entre équivalence et composabilité)
Soit $L : \mathcal{P} \times \mathcal{D} \to \mathcal{D}$ un langage dont les programmes sont composables et $D \subset \mathcal{D}$ un ensemble de données.

- $\forall p, q \in \mathcal{P} \quad \text{si} \quad p \sqsubseteq p' \quad \text{alors} \quad \forall q \in \mathcal{P} \quad p \circ q \sqsubseteq p' \circ q \quad \text{et} \quad q \circ p \sqsubseteq q \circ p'$
- $\forall p, q \in \mathcal{P} \quad \text{si} \quad p \equiv p' \quad \text{alors} \quad \forall q \in \mathcal{P} \quad p \circ q \equiv p' \circ q \quad \text{et} \quad q \circ p \equiv q \circ p'$
- $\forall p, q \in \mathcal{P}_D \quad \text{si} \quad p \equiv_D p' \quad \text{alors} \quad \forall q \in \mathcal{P} \quad p \circ q \equiv_{L(q)^{-1}(D)} p' \circ q \quad \text{et} \quad q \circ p \equiv_D q \circ p'$

4.2 Optimalité.

Dans la mesure où la comparaison de programmes $\preccurlyeq_\mu$ n'est généralement pas antisymétrique — des programmes obtenus par α-conversion (renommage des variables) ont généralement des coûts d'exécution équivalents — il n'y a pas unicité de la « meilleure évaluation partielle ». Elle n'est généralement pas totale non plus, et il nous faut distinguer plusieurs notions d'*optimalité*.

Définition 4.2. (Optimalité)
Soit $(E, \preccurlyeq)$ un ensemble muni d'une relation et $A \subset E$.

- L'ensemble des *éléments minimaux de* A est $\mathrm{Emin}_\preccurlyeq(A) = \{x \in A \mid \forall y \in A \;\; y \preccurlyeq x \;\Rightarrow\; x \preccurlyeq y\}$
- L'ensemble des *plus petits éléments de* A est $\mathrm{Min}_\preccurlyeq(A) = \{x \in A \mid \forall y \in A \;\; x \preccurlyeq y\}$
- L'ensemble des *minorants de* A *dans* E est $\mathrm{Minor}_{E,\preccurlyeq}(A) = \{x \in E \mid \forall y \in A \;\; x \preccurlyeq y\}$
- L'ensemble des *bornes inférieures de* A *dans* E est $\mathrm{Inf}_{E,\preccurlyeq}(A) = \mathrm{Max}_\preccurlyeq(\mathrm{Minor}_{E,\preccurlyeq}(A)) =$
  $\{x \in E \mid \forall y \in A \;\; x \preccurlyeq y \text{ et } \forall z \in E \;\; (\forall y \in A \;\; z \preccurlyeq y) \;\Rightarrow\; z \preccurlyeq x\}$ avec $\mathrm{Max}_\preccurlyeq(A) = \mathrm{Min}_\succcurlyeq(A)$

On définit de même un *élément maximal*, un *plus grand élément*, un *majorant* et une *borne supérieure*. Par ailleurs, on note $\mathrm{Opt}_{\preccurlyeq|A} = \mathrm{Opt}_\preccurlyeq \cap A$ pour chacun des ensembles Opt définis ci-dessus.

Les propriétés élémentaires suivantes permettent de comparer entre elles ces différentes notions d'optimalité sur une partie A d'un ensemble E. Rappelons que le symbole $\approx$ représente l'équivalence $\preccurlyeq \cap \succcurlyeq$.

Proposition 4.3. (Relations entre les optimums)
Soit $(E, \preccurlyeq)$ un ensemble muni d'une relation et $A \subset E$. Alors (avec les notations de la définition 3.1),

(1) $\mathrm{Emin}_{\preccurlyeq|A}(E) \subset \mathrm{Emin}_\preccurlyeq(A) = \mathrm{Min}_{\not\succ}(A) \subset A$
(2) $\mathrm{Min}_{\preccurlyeq|A}(E) \subset \mathrm{Min}_\preccurlyeq(A) \subset^1 \mathrm{Emin}_\preccurlyeq(A) \subset A$
(3) $\mathrm{Minor}_{A,\preccurlyeq}(A) = \mathrm{Minor}_{E,\preccurlyeq|A}(A) = \mathrm{Min}_\preccurlyeq(A)$ et $\mathrm{Minor}_{E,\preccurlyeq|A}(E) = \mathrm{Min}_{\preccurlyeq|A}(E)$
(4) $\mathrm{Inf}_{A,\preccurlyeq}(A) = \mathrm{Inf}_{E,\preccurlyeq|A}(A) = \mathrm{Min}_\preccurlyeq(A)$ et $\mathrm{Inf}_{E,\preccurlyeq|A}(E) = \mathrm{Min}_{\preccurlyeq|A}(E)$
(5) $\mathrm{Emin}_{\preccurlyeq|A}(E) \approx \mathrm{Min}_{\preccurlyeq|A}(E) \approx \mathrm{Min}_\preccurlyeq(A) \approx \mathrm{Emin}_\preccurlyeq(A)$ et $\mathrm{Min}_\preccurlyeq(A) \preccurlyeq A$

De plus, on a (1.) égalité lorsque $\preccurlyeq$ est totale car alors $\preccurlyeq \,=\, \not\succ$.

Au vu des points (3) et (4), il n'existe donc que quatre types différents d'ensembles optimaux qui sont inclus dans A, car les définitions qui emploient Minor et Inf se ramènent toutes à Min. Le tableau 4.1 en donne quelques propriétés supplémentaires, en particulier selon la manière dont la relation $\preccurlyeq$ est construite.

4.3 Optimisation et évaluations partielles.

Comme nous l'avons vu à propos du coût statique (cf. §3.3), comparer les coûts de deux programmes n'a de sens que sur un sous-ensemble commun de leur domaine de définition. C'est pourquoi un coût statique opère sur un support $D \subset \mathcal{D}$ et borne la comparaion des programmes à l'ensemble $\mathcal{P}_D = \{p \in \mathcal{P} \mid D \subset \mathit{Dom}(p)\}$.

Considérons donc $p \in \mathcal{P}$ un programme, $D \subset \mathit{Dom}(p)$ un support, $\preccurlyeq_D$ une comparaison de coût sur $\mathcal{P}_D$ et $P \subset \mathcal{P}_D$ un ensemble de programmes équivalents à p pour une observation $\equiv_D$ sur $\mathcal{P}_D$. Une *optimisation de* p *dans* P *sur* D *selon* $\preccurlyeq_D$ consiste à chercher des (ou les plus) petits éléments de $\{p' \in P \mid p' \preccurlyeq_{|D} p\}$.

Opt	$\mathrm{Min}_{\preccurlyeq}(A)$	$\mathrm{Min}_{\preccurlyeq\|A}(E)$	$\mathrm{Emin}_{\preccurlyeq}(A)$	$\mathrm{Emin}_{\preccurlyeq\|A}(E)$
$\mathrm{Opt}_{\prec} = \mathrm{Opt}_{\preccurlyeq}$	$\varnothing$	$\varnothing$	✓	✓
$\preccurlyeq_1 \subset \preccurlyeq_2 \Rightarrow \mathrm{Opt}_{\preccurlyeq_1} \subset \mathrm{Opt}_{\preccurlyeq_2}$	✓	✓		
$\mathrm{Opt}_{\cap_{i\in I}\preccurlyeq_i} = \bigcap_{i\in I} \mathrm{Opt}_{\preccurlyeq_i}$	✓	✓	$\supset$	$\supset$
$\mathrm{Opt}_{\cup_{i\in I}\preccurlyeq_i} = \bigcup_{i\in I} \mathrm{Opt}_{\preccurlyeq_i}$	$\supset$	$\supset$		
$A \subset A' \Rightarrow \mathrm{Opt}(A) \subset \mathrm{Opt}(A')$		✓		✓
$A \subset A' \Rightarrow \mathrm{Opt}(A) = \mathrm{Opt}(A') \cap A$	$\supset$	✓	$\supset$	✓

Tableau 4.1 : Propriétés des ensembles optimaux sur des relations combinées

4.3.1 Évaluation partielle.

L'évaluation partielle est une optimisation de $p \in \mathcal{P}$ sur un support $D = Dom(p)$ et une classe P de programmes équivalents à p qui vérifie $Equiv_{\mathrm{str}}(p) \subset P \subset Equiv_{\mathrm{par}}(p)$ (on a bien alors $P \subset \mathcal{P}_D$). Si $P = Equiv_{\mathrm{str}}(p)$, on recherche uniquement parmi les programmes strictement équivalents à p. Si $P = Equiv_{\mathrm{par}}(p)$, on s'autorise un choix plus large avec des programmes qui simulent p avec exactitude sur $Dom(p)$, mais qui peuvent avoir un comportement différent par ailleurs.

Il faut noter que D et P sont fixés pendant tout le processus d'optimisation : on ne cherche pas $p' \preccurlyeq_{|Dom(p)} p$ dans $\mathcal{P}_{Dom(p)}$, puis $p'' \preccurlyeq_{|Dom(p')} p'$ dans $\mathcal{P}_{Dom(p')}$, car p et p'' ne sont pas nécessairement comparables. Plus précisément, la seule comparaison qui aurait un sens serait $p'' \preccurlyeq_{|Dom(p)} p$, mais elle n'est pas garantie par les deux comparaisons précédentes.

Définition 4.3. (Évaluation partielle)
Soient $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ un langage, $(\mathcal{C}^D, \preccurlyeq_D, \mu_D)_{D\subset\mathcal{D}}$ un modèle de performance de L, $p \in \mathcal{P}$ un programme, et P un ensemble vérifiant $Equiv_{\mathrm{str}}(p) \subset P \subset Equiv_{\mathrm{par}}(p)$. On note $\preccurlyeq \; = \; \preccurlyeq_{\mathcal{D}|D,\mu_D}$ la comparaison restreinte au support $D = Dom(p)$. L'*ensemble des évaluations partielles de p basées sur P* est défini par :

- $Evap_{P,\preccurlyeq}(p) = \{q \in P \mid q \preccurlyeq p\}$

Nous omettons les indices P et $\preccurlyeq$ lorsque le contexte lève toute ambiguïté. Si la relation de coût porte un indice $\preccurlyeq_{\mathrm{stat}}$, nous notons $Evap_{\mathrm{stat}}(p)$ un ensemble $Evap_{\preccurlyeq_{\mathrm{stat}}}(p)$.

- L'évaluation partielle *stricte* est basée sur $P = Equiv_{\mathrm{str}}(p)$: on a $Evap_{\mathrm{str}}(p) = Evap_{Equiv_{\mathrm{str}}(p),\preccurlyeq}(p)$.
- L'évaluation partielle *paresseuse* est basée sur $P = Equiv_{\mathrm{par}}(p)$: on a $Evap_{\mathrm{par}}(p) = Evap_{Equiv_{\mathrm{par}}(p),\preccurlyeq}(p)$.

Par abus de langage, on appelle *une* évaluation partielle *un* élément de *l'*évaluation partielle $Evap(p)$. Lorsque $Evap(p)$ est vide, on dit que *p n'a pas d'évaluation partielle*.

Une *évaluation partielle* est donc[1] un programme qui est à la fois équivalent (au sens de P) et meilleur (au sens de $\preccurlyeq$) que p. Le choix de P détermine si l'on admet ou non des programmes « paresseux », de domaine éventuellement plus grand que $Dom(p)$.

[1] Le terme d'« évaluation partielle » n'a plus vraiment de sens dans ce contexte, et l'on attendrait plutôt « optimisation ». Dans le langage courant, une *évaluation partielle de p* est un programme p' obtenu par transformation de p selon. . . des techniques d'évaluation partielle.

4.3.2 Meilleures évaluations partielles.

On déduit des généralités sur l'optimalité (cf. §4.2) une définition des *évaluations partielles optimales et minimales*. L'ensemble de base est ici $E = P$, un ensemble de programmes équivalents en un certain sens à p, et l'on s'intéresse à un sous-ensemble $A = Evap_{P,\preccurlyeq}(p)$. Nous employons a priori les quatre types différents d'optimalité ; nous montrerons qu'elles se traduisent par uniquement trois types de meilleures évaluations partielles.

Définition 4.4. (Évaluation partielle optimale, minimale)
Reprenons les notations de la définition 4.3. L'*ensemble des évaluations partielles optimales* (resp. *minimales*) *fortes* (resp. *faibles*) *de p basées sur P* est défini par :

- $Evap^{\mathrm{optF}}_{P,\preccurlyeq}(p) = \mathrm{Min}_{\preccurlyeq|Evap_{P,\preccurlyeq}(p)}(P) = \{q \in P \mid q \preccurlyeq p \text{ et } \forall q' \in P \quad q \preccurlyeq q'\}$
- $Evap^{\mathrm{optf}}_{P,\preccurlyeq}(p) = \mathrm{Min}_{\preccurlyeq}(Evap_{P,\preccurlyeq}(p)) = \{q \in P \mid q \preccurlyeq p \text{ et } \forall q' \in P \quad q' \preccurlyeq p \Rightarrow q \preccurlyeq q'\}$
- $Evap^{\mathrm{minF}}_{P,\preccurlyeq}(p) = \mathrm{Emin}_{\preccurlyeq|Evap_{P,\preccurlyeq}(p)}(P) = \{q \in P \mid q \preccurlyeq p \text{ et } \forall q' \in P \quad q' \preccurlyeq q \Rightarrow q \preccurlyeq q'\}$
- $Evap^{\mathrm{minf}}_{P,\preccurlyeq}(p) = \mathrm{Emin}_{\preccurlyeq}(Evap_{P,\preccurlyeq}(p)) = \{q \in P \mid q \preccurlyeq p \text{ et } \forall q' \in P \quad q' \preccurlyeq p \wedge q' \preccurlyeq q \Rightarrow q \preccurlyeq q'\}$

Nous employons également les mêmes conventions qu'à la définition 4.3 en ce qui concerne les indices.

Il y a ainsi plusieurs notions de « *meilleure évaluation partielle de p dans P* » :

- Une évaluation partielle *optimale forte* est meilleure que tous les autres programmes de P. C'est également (prop. 4.3, points 3 et 4) une évaluation partielle qui est une borne inférieure (resp. un minorant) de P.
- Une évaluation partielle *optimale faible* est meilleure que les programmes de P qui sont eux-mêmes meilleurs que p. Autrement dit, elle est meilleure que toutes les autres évaluations partielles. C'est également (prop. 4.3, points 3 et 4) une évaluation partielle qui est une borne inférieure (resp. un minorant) de l'ensemble des évaluations partielles.
- Une évaluation partielle *minimale forte* est telle qu'il n'existe pas dans P de programme strictement meilleur. Autrement dit, il n'existe pas de programme équivalent à p qui soit strictement meilleur.
- Une évaluation partielle *minimale faible* est telle qu'il n'existe pas de programme strictement meilleur parmi ceux de P qui sont aussi meilleurs que p. Autrement dit, aucune évaluation partielle n'est strictement meilleure.

Comme pour l'évaluation partielle simple, le choix de P détermine là encore si l'on admet ou non des programmes « paresseux ».

Proposition 4.4. (Relations entre les évaluations partielles)
Avec les notations de la définition 4.4, on a :

(1) $Evap(p) \neq \varnothing$ si $\preccurlyeq$ est réflexive car alors $p \in Evap(p)$
(2) $Evap^{\mathrm{optF}}_{\preccurlyeq}(p) = \bigcap_{q \in P} Evap_{\preccurlyeq}(q)$
(3) $Evap^{\mathrm{optf}}_{\preccurlyeq}(p) = Evap_{\preccurlyeq}(p) \cap (\bigcap_{q \in Evap_{\preccurlyeq}(p)} Evap_{\preccurlyeq}(q)) =^1 \bigcap_{q \in Evap_{\preccurlyeq}(p)} Evap_{\preccurlyeq}(q)$
(4) $Evap^{\mathrm{minF}}_{\preccurlyeq}(p) = Evap_{\preccurlyeq}(p) \cap (\bigcap_{q \in P} Evap_{\not\succcurlyeq}(q))$
(5) $Evap^{\mathrm{minf}}_{\preccurlyeq}(p) = Evap_{\preccurlyeq}(p) \cap (\bigcap_{q \in Evap_{\preccurlyeq}(p)} Evap_{\not\succcurlyeq}(q))$
(6) $Evap^{\mathrm{optF}}(p) \subset^2 Evap^{\mathrm{optf}}(p) \subset^2 Evap^{\mathrm{minF}}(p) = Evap^{\mathrm{minf}}(p) \subset Evap(p) \subset P$
(7) $Evap^{\mathrm{optF}}(p) \approx Evap^{\mathrm{optf}}(p) \approx Evap^{\mathrm{minF}}(p) \approx Evap^{\mathrm{minf}}(p) \preccurlyeq Evap(p)$
(8) Si $L : \mathcal{P} \to (\mathcal{D} \to \mathcal{R})$ est injective et $\preccurlyeq$ est réflexive alors
$Evap^{\mathrm{optF}}_{\mathrm{str}}(p) = Evap^{\mathrm{optf}}_{\mathrm{str}}(p) = Evap^{\mathrm{min}}_{\mathrm{str}}(p) = Evap_{\mathrm{str}}(p) = Equiv_{\mathrm{str}}(p) = \{p\}$

On a égalité (1.) si $Evap(p)$ est non-vide / (2.) si $\preccurlyeq$ est totale. Les inclusions sont strictes dans le cas général.

Il est important de noter que (pt. 2) l'évaluation partielle optimale forte ne dépend pas de p et de son exécution, mais uniquement de la fonction mathématique $L(p)$. C'est en quelque sorte les meilleurs programmes dont la sémantique est $L(p)$ (au sens de P).

D'autre part (pt. 6), les évaluations partielles minimales forte et faible coïncident ; nous notons alors simplement $Evap^{\min}_{\preccurlyeq,P}(p)$. Il n'existe donc que trois types différents de « meilleure évaluation partielle », qui en outre sont confondus lorsque $\preccurlyeq$ est totale (nous notons en ce cas $Evap^{\mathrm{opt}}_{\preccurlyeq,P}(p)$ l'évaluation partielle optimale ou minimale), ce qui cependant est relativement rare. Même lorsque $\preccurlyeq$ n'est pas totale, les différentes évaluations partielles optimales et minimales ne diffèrent pas substanciellement car tous leurs programmes sont équivalents entre eux.

Le tableau 4.2 donne quelques identités sur les ensembles d'évaluations partielles, et en particulier celles qui concernent les évaluations partielles construites sur des coûts combinés. Remarquons à ce sujet que le produit $\preccurlyeq_1 \otimes \preccurlyeq_2$ de relations sur $\mathcal{C}_1 \times \mathcal{C}_2$ n'a pas sa place dans ce tableau car il est « transformé » en une intersection $\preccurlyeq_{1,\mu_1} \cap \preccurlyeq_{2,\mu_2}$ sur P par une mesure de performance (déf. 3.7). Le cas est similaire pour le produit lexicographique (déf. 3.8).

$Evap^{\mathrm{xxx}}_{P,\preccurlyeq}(p)$	$Evap$	$Evap^{\min}$	$Evap^{\mathrm{optf}}$	$Evap^{\mathrm{optF}}$
$Evap_{\prec} = Evap_{\preccurlyeq}$	$\subset$	✓	$\varnothing$	$\varnothing$
$\preccurlyeq_1 \subset \preccurlyeq_2 \;\Rightarrow\; Evap_{\preccurlyeq_1} \subset Evap_{\preccurlyeq_2}$	✓		✓	✓
$Evap_{\cap_{i\in I}\preccurlyeq_i} = \bigcap_{i\in I} Evap_{\preccurlyeq_i}$	✓	$\supset$	✓	✓
$Evap_{\cup_{i\in I}\preccurlyeq_i} = \bigcup_{i\in I} Evap_{\preccurlyeq_i}$	✓		$\supset$	$\supset$
$P \subset P' \;\Rightarrow\; Evap_P \subset Evap_{P'}$	✓	✓		✓
$P \subset P' \;\Rightarrow\; Evap_P = Evap_{P'} \cap P$	✓	✓	$\supset$	✓
$p \equiv p' \;\wedge\; p \preccurlyeq p' \;\Rightarrow\; Evap(p) = Evap(p')$	$\subset$	$\subset$	$\supset$	✓

Tableau 4.2 : Identités sur les ensembles d'évaluations partielles

La section suivante met en valeur ces résultats sur quelques comparaisons de coûts définies au chapitre précédent.

4.4 Évaluation partielle et mesure de coût.

Les résultats qui concernent les comparaisons de coût (cf. §3.6) et les évaluations partielles (cf. §4.3) permettent des jugements à la fois qualitatifs et quantitatifs. Nous l'illustrons sur quelques coûts statiques qui ont été définis à la section §3.4.

Soient les programmes nand$i$ de BOOL suivants.

```
nand1(X,Y) = if X then (if Y then false else true) else true
nand2(X,Y) = if Y then (if X then false else true) else true
```

```
nand3(X,Y) = if X then (if Y then false else X    ) else true
nand4(X,Y) = if Y then (if X then false else Y    ) else true
nand5(X,Y) = if X then (if Y then false else true)
                   else (if Y then true  else true)
nand6(X,Y) = if Y then (if X then false else true)
                   else (if X then true  else true)
```

Ces programmes vérifient les équivalences `nand1` $\equiv \ldots \equiv$ `nand6`. Par conséquent, les ensembles de programmes strictement équivalents sont :

$$\mathit{Equiv}_{\mathrm{str}}(\mathtt{nand1}) = \ldots = \mathit{Equiv}_{\mathrm{str}}(\mathtt{nand6}) = \{\mathtt{nand1}, \mathtt{nand2}, \mathtt{nand3}, \mathtt{nand4}, \mathtt{nand5}, \mathtt{nand6}, \ldots\}$$

Les programmes `nand`i sont définis sur tout Bool $\times$ Bool. On a donc $\mathit{Equiv}_{\mathrm{str}}(\mathtt{nand}i) = \mathit{Equiv}_{\mathrm{par}}(\mathtt{nand}i)$. Notons qu'il n'y a que six autres programmes équivalents aux `nand`i et ne comportant que deux `if` imbriqués (et trois `if` en tout) :

```
nand7 (X,Y) = if X then (if Y then false else X    ) else (if Y then true  else true)
nand8 (X,Y) = if Y then (if X then false else Y    ) else (if X then true  else true)
nand9 (X,Y) = if X then (if Y then false else true) else (if Y then Y     else true)
nand10(X,Y) = if Y then (if X then false else true) else (if X then X     else true)
nand11(X,Y) = if X then (if Y then false else X    ) else (if Y then Y     else true)
nand12(X,Y) = if Y then (if X then false else Y    ) else (if X then X     else true)
```

Les mesures de coût suivantes leur sont associées :

$$
\begin{array}{rcl}
\mu(\mathtt{nand1}\,(X,Y)) & = & \left\{\begin{array}{lll} \text{si } X = \mathit{true} & \text{alors} & 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \\ \text{si } X = \mathit{false} & \text{alors} & c_{\mathtt{if}} + c_{var} + c_{bool} \end{array}\right. \\
\mu(\mathtt{nand2}\,(X,Y)) & = & \left\{\begin{array}{lll} \text{si } Y = \mathit{true} & \text{alors} & 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \\ \text{si } Y = \mathit{false} & \text{alors} & c_{\mathtt{if}} + c_{var} + c_{bool} \end{array}\right. \\
\mu(\mathtt{nand3}\,(X,Y)) & = & \left\{\begin{array}{lll} \text{si } X = \mathit{true} \text{ et } Y = \mathit{true} & \text{alors} & 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \\ \text{si } X = \mathit{true} \text{ et } Y = \mathit{false} & \text{alors} & 2\,c_{\mathtt{if}} + 3\,c_{var} \\ \text{si } X = \mathit{false} & \text{alors} & c_{\mathtt{if}} + c_{var} + c_{bool} \end{array}\right. \\
\mu(\mathtt{nand4}\,(X,Y)) & = & \left\{\begin{array}{lll} \text{si } Y = \mathit{true} \text{ et } X = \mathit{true} & \text{alors} & 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \\ \text{si } Y = \mathit{true} \text{ et } X = \mathit{false} & \text{alors} & 2\,c_{\mathtt{if}} + 3\,c_{var} \\ \text{si } Y = \mathit{false} & \text{alors} & c_{\mathtt{if}} + c_{var} + c_{bool} \end{array}\right. \\
\mu(\mathtt{nand5}\,(X,Y)) & = & 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \\
\mu(\mathtt{nand6}\,(X,Y)) & = & 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool}
\end{array}
$$

Rappelons (cf. §3.2.5, domaine de coût implicite) que les coûts élémentaires vérifient :

$$0 \prec c_{bool} \prec c_{var} \preccurlyeq c_{\mathtt{if}} \prec c_{fun_n} \prec c_{fun_m} \quad \text{pour} \quad 0 \preccurlyeq n < m$$

4.4.1 Évaluation partielle et coût absolu.

Nous avons défini (cf. §3.4.1) la comparaison absolue sur un support D comme :

- $\forall p, q \in \mathcal{P}_D \quad p \preccurlyeq_{\mathrm{abs}} q \quad \text{ssi} \quad \forall d \in D \quad \mu(p\ d) \preccurlyeq \mu(q\ d)$

Avec $D = \text{Bool} \times \text{Bool}$ comme support, les programmes $\texttt{nand}i$ se comparent ainsi.

$$\texttt{nand3} \succ_{\text{abs}} \texttt{nand1} \prec_{\text{abs}} \texttt{nand5} \approx_{\text{abs}} \texttt{nand6} \succ_{\text{abs}} \texttt{nand2} \prec_{\text{abs}} \texttt{nand4}$$

Ce sont les seules relations de coût absolu entre les programmes $\texttt{nand}i$. D'autre part, il est clair qu'un programme `nand` équivalent doit comporter au minimum deux `if` emboîtés. Les expressions terminales de ces `if` ont un coût minimum de c_{bool}. Sachant qu'il est plus coûteux de rajouter un troisième `if` que d'employer une variable, puisque $c_{var} \preccurlyeq c_{\texttt{if}}$, et superflu de faire un appel fonctionnel, nous en déduisons :

$$\begin{array}{rcl}
Evap_{\text{abs}}(\texttt{nand1}) &=& \{\texttt{nand1}\}\\
Evap_{\text{abs}}(\texttt{nand2}) &=& \{\texttt{nand2}\}\\
Evap_{\text{abs}}(\texttt{nand3}) &=& \{\texttt{nand1},\texttt{nand3}\}\\
Evap_{\text{abs}}(\texttt{nand4}) &=& \{\texttt{nand2},\texttt{nand4}\}\\
Evap_{\text{abs}}(\texttt{nand5}) &=& \{\texttt{nand1},\texttt{nand2},\texttt{nand5},\texttt{nand6}\}\\
Evap_{\text{abs}}(\texttt{nand6}) &=& \{\texttt{nand1},\texttt{nand2},\texttt{nand5},\texttt{nand6}\}
\end{array}$$

Il est très rare qu'il existe des évaluations partielles optimales fortes ; leur ensemble est d'ailleurs inclus dans tous les autres types de meilleures évaluations partielles (prop. 4.4.6). Or $\preccurlyeq_{\text{abs}}$ est la relation la plus forte que nous ayons rencontrée, puisqu'elle aussi est incluse dans toutes les autres, ou plus précisément dans toutes celles qui sont stables (cf. §3.5.1), ce qui est le cas de la majorité des comparaisons de coût (cf. §3.6). Comme l'indique la propriété « $\preccurlyeq_1 \subset \preccurlyeq_2 \Rightarrow Evap^{\text{optF}}_{\preccurlyeq_1}(p) \subset Evap^{\text{optF}}_{\preccurlyeq_2}(p)$ » (tabl. 4.2), il est donc encore plus rare d'en trouver pour le coût absolu. Effectivement, les programmes $\texttt{nand}i$ n'ont pas d'évaluation partielle optimale forte, car les deux programmes `nand1` et `nand2` ont des évaluations partielles disjointes :

$$Evap^{\text{optF}}_{\text{abs}}(\texttt{nand}i) = \bigcap_{p\equiv\texttt{nand}i} Evap_{\text{abs}}(p) = \varnothing$$

Certains programmes ont néanmoins une évaluation partielle optimale faible. On la calcule par la formule $Evap^{\text{optf}}_{\text{abs}}(\texttt{nand}i) = \bigcap_{q\in Evap_{\text{abs}}(\texttt{nand}i)} Evap_{\text{abs}}(q)$.

$$\begin{array}{rcl}
Evap^{\text{optf}}_{\text{abs}}(\texttt{nand1}) = Evap^{\text{optf}}_{\text{abs}}(\texttt{nand3}) &=& \{\texttt{nand1}\}\\
Evap^{\text{optf}}_{\text{abs}}(\texttt{nand2}) = Evap^{\text{optf}}_{\text{abs}}(\texttt{nand4}) &=& \{\texttt{nand2}\}\\
Evap^{\text{optf}}_{\text{abs}}(\texttt{nand5}) = Evap^{\text{optf}}_{\text{abs}}(\texttt{nand6}) &=& \varnothing
\end{array}$$

Les programmes `nand5` et `nand6` n'ont pas d'évaluation partielle optimale faible pour la même raison qu'ils n'avaient pas d'évaluation partielle optimale forte : `nand1` et `nand2`, qui apparaissent dans $Evap_{\text{abs}}(\texttt{nand5})$ et $Evap_{\text{abs}}(\texttt{nand6})$, ont des évaluations partielles disjointes. Ils ont toutefois une évaluation partielle minimale, pour laquelle il nous faut calculer tout d'abord les évaluations partielles selon $\not\succ$.

$$\begin{array}{rcl}
Evap_{\not\succ_{\text{abs}}}(\texttt{nand1}) &=& \{\texttt{nand1},\texttt{nand2},\texttt{nand4},\ldots\}\\
Evap_{\not\succ_{\text{abs}}}(\texttt{nand2}) &=& \{\texttt{nand1},\texttt{nand2},\texttt{nand3},\ldots\}\\
Evap_{\not\succ_{\text{abs}}}(\texttt{nand3}) = \ldots = Evap_{\not\succ_{\text{abs}}}(\texttt{nand6}) &=& \{\texttt{nand1},\ldots,\texttt{nand6},\ldots\}
\end{array}$$

Les points de suspension indiquent que des programmes autres que `nand1` à `nand6` sont dans l'évaluation partielle.

L'évaluation partielle minimale est $Evap^{\min}_{\text{abs}}(\texttt{nand}i) = Evap_{\text{abs}}(\texttt{nand}i) \cap (\bigcap_{q\in Evap_{\text{abs}}(\texttt{nand}i)} Evap_{\not\succ_{\text{abs}}}(q))$.

$$\begin{array}{rcl}
Evap^{\min}_{\text{abs}}(\texttt{nand1}) = Evap^{\min}_{\text{abs}}(\texttt{nand3}) &=& \{\texttt{nand1}\}\\
Evap^{\min}_{\text{abs}}(\texttt{nand2}) = Evap^{\min}_{\text{abs}}(\texttt{nand4}) &=& \{\texttt{nand2}\}\\
Evap^{\min}_{\text{abs}}(\texttt{nand5}) = Evap^{\min}_{\text{abs}}(\texttt{nand6}) &=& \{\texttt{nand1},\texttt{nand2}\}
\end{array}$$

Il est naturel que des inclusions $Evap^{\mathrm{optF}}_{\mathrm{abs}}(\mathtt{nand}i) \subset Evap^{\mathrm{optf}}_{\mathrm{abs}}(\mathtt{nand}i) \subset Evap^{\mathrm{min}}_{\mathrm{abs}}(\mathtt{nand}i)$ soient en général strictes car le coût $\preccurlyeq_{\mathrm{abs}}$ n'est pas total, même si $\preccurlyeq$ l'est (tabl. 3.3).

En fait, les programmes nand1 et nand2 sont les deux seuls programmes minimaux de $P = Equiv(\mathtt{nand}i)$: il n'y a pas de programme $nand$ équivalent aux nand$i$ de coût strictement inférieur à $\mu(\mathtt{nand1})$ ou $\mu(\mathtt{nand2})$. À ce titre, pour tout programme $nand$ dans P, ce sont les seuls éléments possibles de l'évaluation partielle minimale $Evap^{\mathrm{min}}_{\mathrm{abs}}(nand)$, et donc aussi des évaluations partielles optimales. En outre, puisque nand1 et nand2 ont des évaluation partielles disjointes, pour tout $nand$ on a $Evap^{\mathrm{optF}}_{\mathrm{abs}}(nand) = \varnothing$. C'est naturel car on a vu que $Evap^{\mathrm{optF}}(p)$ ne dépend pas de p (prop. 4.4.2), et donc que $Evap^{\mathrm{optF}}_{\mathrm{abs}}(\mathtt{nand}i) = \varnothing$.

4.4.2 Évaluation partielle et coût moyen.

Nous avons vu (cf. §3.6.6) que tous les coûts moyens sont équivalents sur un langage de domaines finis comme BOOL. La mesure de coût moyen explicite (cf. §3.4.4) est définie comme :

- $\forall p \in \mathcal{P} \quad \mu_{\mathrm{moy}}(p) = \sum_{d \in Dom(p)} \mu(p\ d)/|Dom(p)|$
- $\forall p, q \in \mathcal{P} \quad p \preccurlyeq_{\mathrm{moy}} q \quad \text{ssi} \quad \mu_{\mathrm{moy}}(p) \preccurlyeq \mu_{\mathrm{moy}}(q)$

Le coût moyen des programmes nand$i$ est :

$$\begin{aligned}
\mu_{\mathrm{moy}}(\mathtt{nand1}) = \mu_{\mathrm{moy}}(\mathtt{nand2}) &= 3/2\ c_{\mathtt{if}} + 3/2\ c_{var} + c_{bool} \\
\mu_{\mathrm{moy}}(\mathtt{nand3}) = \mu_{\mathrm{moy}}(\mathtt{nand4}) &= 3/2\ c_{\mathtt{if}} + 7/4\ c_{var} + 3/4\ c_{bool} \\
\mu_{\mathrm{moy}}(\mathtt{nand5}) = \mu_{\mathrm{moy}}(\mathtt{nand6}) &= 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool}
\end{aligned}$$

À la différence du coût absolu, tous les programmes nand$i$ sont comparables en moyenne. C'est aussi le cas de tout autre programme $nand$ à la condition que $\preccurlyeq$ soit totale (nous n'avons jusqu'ici pris comme hypothèse que les relations individuelles sur les coûts élémentaires) puisque la construction du coût moyen sur les domaines finis transporte sur le coût statique le fait que la relation de coût dynamique est totale (tabl. 3.3).

$$\mathtt{nand1} \approx_{\mathrm{moy}} \mathtt{nand2} \prec_{\mathrm{moy}} \mathtt{nand3} \approx_{\mathrm{moy}} \mathtt{nand4} \prec_{\mathrm{moy}} \mathtt{nand5} \approx_{\mathrm{moy}} \mathtt{nand6}$$

Il n'est pas étonnant non plus que la hiérarchie relative (cf. §4.4.1) ne soit pas modifiée puisque $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\mathrm{moy}}$ (tabl. 3.3). D'autre part, dans la mesure où l'augmentation du nombre de if multiplie également le nombre de feuilles d'une expression, on peut voir que tout programme équivalent comportant au moins trois if imbriqués a un coût moyen trop important pour figurer dans $Evap_{\mathrm{moy}}(\mathtt{nand}i)$. Une énumération de ceux qui ne comportent que deux if conduit aux évaluations partielles suivantes :

$$\begin{aligned}
Evap_{\mathrm{moy}}(\mathtt{nand1}) = Evap_{\mathrm{moy}}(\mathtt{nand2}) &= \{\mathtt{nand1}, \mathtt{nand2}\} \\
Evap_{\mathrm{moy}}(\mathtt{nand3}) = Evap_{\mathrm{moy}}(\mathtt{nand4}) &= \{\mathtt{nand1}, \mathtt{nand2}, \mathtt{nand3}, \mathtt{nand4}\} \\
Evap_{\mathrm{moy}}(\mathtt{nand5}) = Evap_{\mathrm{moy}}(\mathtt{nand6}) &= \{\mathtt{nand1}, \mathtt{nand2}, \mathtt{nand3}, \mathtt{nand4}, \mathtt{nand5}, \mathtt{nand6}\}
\end{aligned}$$

Les évaluations partielles pour le coût moyen contiennent bien celles pour le coût absolu, puisque $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\mathrm{moy}}$ (tabl. 4.2). Cette fois-ci, tous les programmes nand$i$ admettent des évaluations partielles optimales fortes.

$$Evap^{\mathrm{optF}}_{\mathrm{moy}}(\mathtt{nand}i) = \bigcap_{p \equiv \mathtt{nand}i} Evap_{\mathrm{moy}}(p) = \{\mathtt{nand1}, \mathtt{nand2}\}$$

Il n'est pas suprenant que les programmes aient tous la même évaluation partielle optimale forte, puisque nous avons vu qu'elle ne dépendait que de la base $P = Equiv(\mathtt{nand}i)$, et non du programme nand$i$ considéré.

D'autre part, puisque tous les coûts moyens sont comparables (ce qui a le même effet que si $\preccurlyeq$ était par hypothèse totale), les évaluations partielles optimales fortes, faibles, et minimales coïncident.

$$Evap^{\mathrm{min}}_{\mathrm{moy}}(\mathtt{nand}i) = Evap^{\mathrm{optf}}_{\mathrm{moy}}(\mathtt{nand}i) = Evap^{\mathrm{optF}}_{\mathrm{moy}}(\mathtt{nand}i) = \{\mathtt{nand1}, \mathtt{nand2}\}$$

Notons également que l'évaluation partielle optimale, même forte, n'est pas nécessairement unique.

4.4.3 Évaluation partielle et coût maximum.

Comme nous l'avons signalé (cf. §3.4.8), il n'est pas très intéressant d'employer le coût maximum « nature ». Il faut le composer avec un autre coût statique : $\preccurlyeq_{\mathrm{stat}} \cap \preccurlyeq_{\max}$. Cependant, déterminer les ensembles d'évaluations partielles propres du coût maximum est tout à fait justifié puisque les propriétés de compatibilité avec l'intersection (tabl. 4.2),

- $Evap_{\preccurlyeq_1 \cap \preccurlyeq_2}(p) = Evap_{\preccurlyeq_1}(p) \cap Evap_{\preccurlyeq_1}(p)$
- $Evap^{\mathrm{min}}_{\preccurlyeq_1 \cap \preccurlyeq_2}(p) \supset Evap^{\mathrm{min}}_{\preccurlyeq_1}(p) \cap Evap^{\mathrm{min}}_{\preccurlyeq_2}(p)$
- $Evap^{\mathrm{optf}}_{\preccurlyeq_1 \cap \preccurlyeq_2}(p) = Evap^{\mathrm{optf}}_{\preccurlyeq_1}(p) \cap Evap^{\mathrm{optf}}_{\preccurlyeq_2}(p)$
- $Evap^{\mathrm{optF}}_{\preccurlyeq_1 \cap \preccurlyeq_2}(p) = Evap^{\mathrm{optF}}_{\preccurlyeq_1}(p) \cap Evap^{\mathrm{optF}}_{\preccurlyeq_2}(p)$

permettent de calculer, ou d'approcher, les évaluations partielles de $\preccurlyeq_{\mathrm{stat}} \cap \preccurlyeq_{\max}$ à partir des évaluations partielles individuelles de $\preccurlyeq_{\mathrm{stat}}$ et $\preccurlyeq_{\max}$.

Le coût maximum dans le cas d'un domaine fini non-vide et d'une relation $\preccurlyeq$ que nous supposons ici totale est :

- $\forall p \in \mathcal{P} \quad \mu_{\max}(p) = \max_{d \in Dom(p)}(\mu(p\ d))$
- $\forall p, q \in \mathcal{P} \quad p \preccurlyeq_{\max} q \quad \text{ssi} \quad \mu_{\max}(p) \preccurlyeq \mu_{\max}(q)$

Le coût maximum des programmes $\mathtt{nand}i$ est :

$$\begin{aligned}
\mu_{\max}(\mathtt{nand1}) = \mu_{\max}(\mathtt{nand2}) &= 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \\
\mu_{\max}(\mathtt{nand3}) = \mu_{\max}(\mathtt{nand4}) &= 2\,c_{\mathtt{if}} + 3\,c_{var} \\
\mu_{\max}(\mathtt{nand5}) = \mu_{\max}(\mathtt{nand6}) &= 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool}
\end{aligned}$$

La hiérarchie reste bien sûr compatible avec le coût absolu (cf. §4.4.1) car, puisque $\preccurlyeq$ est totale et le domaine fini, on a $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\max}$ (tabl. 3.3). Elle est en revanche différente de celle du coût moyen (cf. §4.4.2).

$$\mathtt{nand1} \approx_{\max} \mathtt{nand2} \approx_{\max} \mathtt{nand5} \approx_{\max} \mathtt{nand6} \prec_{\max} \mathtt{nand3} \approx_{\max} \mathtt{nand4}$$

Là encore, le nombre de `if` emboîtés limite la recherche des évaluations partielles.

$$\begin{aligned}
Evap_{\max}(\mathtt{nand1}) = Evap_{\max}(\mathtt{nand2}) &= \{\mathtt{nand1}, \mathtt{nand2}, \mathtt{nand5}, \mathtt{nand6}\} \\
Evap_{\max}(\mathtt{nand3}) = Evap_{\max}(\mathtt{nand4}) &= \{\mathtt{nand1}, \ldots, \mathtt{nand6}, \mathtt{nand7}, \ldots, \mathtt{nand12}\} \\
Evap_{\max}(\mathtt{nand5}) = Evap_{\max}(\mathtt{nand6}) &= \{\mathtt{nand1}, \mathtt{nand2}, \mathtt{nand5}, \mathtt{nand6}\}
\end{aligned}$$

Les programmes `nand7` à `nand12` (cf. §4.4) sont les seuls autres programmes équivalents à `nand` ne comportant que deux `if` imbriqués. La compatibilité de l'évaluation partielle sur l'intersection nous permet d'en déduire les évaluations partielles pour $\preccurlyeq_{\text{moy-max}} = \preccurlyeq_{\mathrm{moy}} \cap \preccurlyeq_{\max}$.

$$\begin{aligned}
Evap_{\text{moy-max}}(\mathtt{nand1}) = Evap_{\text{moy-max}}(\mathtt{nand2}) &= \{\mathtt{nand1}, \mathtt{nand2}\} \\
Evap_{\text{moy-max}}(\mathtt{nand3}) = Evap_{\text{moy-max}}(\mathtt{nand4}) &= \{\mathtt{nand1}, \mathtt{nand2}, \mathtt{nand3}, \mathtt{nand4}\} \\
Evap_{\text{moy-max}}(\mathtt{nand5}) = Evap_{\text{moy-max}}(\mathtt{nand6}) &= \{\mathtt{nand1}, \mathtt{nand2}, \mathtt{nand5}, \mathtt{nand6}\}
\end{aligned}$$

Puisque ce coût maximum est total (tabl. 3.3), évaluations partielles optimales fortes, faibles, et minimales coïncident à nouveau.

$$Evap^{\min}_{\max}(\texttt{nand}i) = Evap^{\mathrm{opt}}_{\max}(\texttt{nand}i) = \{\texttt{nand1}, \texttt{nand2}, \texttt{nand5}, \texttt{nand6}\}$$

Grâce à la compatibilité avec l'intersection, on déduit :

$$Evap^{\mathrm{optF}}_{\text{moy-max}}(\texttt{nand}i) = Evap^{\mathrm{optF}}_{\mathrm{moy}}(\texttt{nand}i) \cap Evap^{\mathrm{optF}}_{\max}(\texttt{nand}i) = \{\texttt{nand1}, \texttt{nand2}\}$$

Dans cet exemple, bien que $\preccurlyeq_{\mathrm{moy}} \not\subset \preccurlyeq_{\max}$ et $\preccurlyeq_{\mathrm{moy}} \not\supset \preccurlyeq_{\max}$ l'évaluation partielle optimale en moyenne des programmes $\texttt{nand}i$ est en fait également optimale pour le coût maximum : $Evap^{\mathrm{opt}}_{\mathrm{moy}}(\texttt{nand}i) \subset Evap^{\mathrm{opt}}_{\max}(\texttt{nand}i)$.

4.5 Difficultés intrinsèques de l'évaluation partielle optimale.

L'indécidabilité de l'équivalence des programmes ne permet généralement pas de connaître complètement l'ensemble $Evap_{\preccurlyeq}(p)$ et, à plus forte raison, pas non plus les ensembles $Evap^{\min}_{\preccurlyeq}(p)$, $Evap^{\mathrm{optf}}_{\preccurlyeq}(p)$ et $Evap^{\mathrm{optF}}_{\preccurlyeq}(p)$. Toutefois, les cas d'existence d'évaluations partielles optimales ou minimales sont rares ; ces ensembles sont le plus souvent vides.

Il faut noter la différence entre le problème de trouver un meilleur programme pour un modèle d'exécution fixé, et celui de trouver une meilleure compilation pour une machine d'exécution donnée (cf. §4.7), comme c'est le cas en particulier dans les recherches d'une meilleure stratégie de réduction lorsque la sémantique est donnée par un procédé non-déterministe [BL79, Lév80, AL93].

Une partie des résultats présentés ici est reprise et complétée dans la conclusion, dans le rubrique qui traite des limites de l'évaluation partielle.

4.5.1 Conditions d'inexistence.

Parce que peu de programmes sont comparables entre eux, les évaluations partielles optimales sont le plus souvent vides. En ce qui concerne l'évaluation partielle minimale, il y a deux tendances opposées : si peu de programmes sont comparables, il est plus facile de trouver des programmes tels qu'il n'en existe pas d'autres de coût strictement inférieur, mais d'un autre coté il y a aussi moins d'évaluations partielles.

Évaluation partielle optimale.

Les cas courants pour lesquels on peut décider qu'il n'y a pas d'évaluation partielle optimale sont déduits des points (2) et (3) de la proposition 4.4 :

- $Evap^{\mathrm{optF}}_{\preccurlyeq}(p) = \varnothing$ si $\exists\, q_1 \neq q_2 \in P$ tq $Evap_{\preccurlyeq}(q_1) \cap Evap_{\preccurlyeq}(q_2) = \varnothing$
- $Evap^{\mathrm{optf}}_{\preccurlyeq}(p) = \varnothing$ si $\exists\, q_1 \neq q_2 \in P$ tq $q_1, q_2 \preccurlyeq p$ et $Evap_{\preccurlyeq}(q_1) \cap Evap_{\preccurlyeq}(q_2) = \varnothing$

C'est parce que les programmes `nand1` et `nand2` ont des évaluations partielles disjointes pour le coût absolu qu'ils n'ont pas d'évaluation partielle optimale (cf. §4.4.1). En pratique, il faut exhiber ces deux programmes q_1 et q_2 pour affirmer qu'il n'y a pas d'évaluation partielle optimale forte ou faible.

Évaluation partielle minimale.

Deux cas peuvent se présenter dans l'étude de l'évaluation partielle minimale $Evap^{\min}(p_0)$: ou bien il n'y a pas de programme de coût strictement inférieur à p_0, auquel cas p_0 est minimal et $p_0 \in Evap^{\min}(p_0)$; ou bien il existe un programme p_1 strictement meilleur que p_0, et l'on peut itérer le processus et rechercher s'il existe un programme p_2 strictement meilleur que p_1, etc. Il n'y a pas d'évaluation partielle minimale, ni par conséquent d'évaluation partielle optimale, lorsque ce processus est sans fin, c'est-à-dire lorsque l'ensemble des programmes n'est pas bien fondé pour la relation de coût statique $\prec$. Considérons par exemple la famille de programmes SML $(\texttt{length}_n)_{n\in\mathbb{N}}$ définie par :

```
fun lengthₙ(L₀) =
  case L₀ of nil => 0 | _::L₁ => (1 +
   ...
     case Lₙ of  nil => 0 | _::Lₙ₊₁ => (1 + lengthₙ(Lₙ₊₁))...)
```

La fonction $\texttt{length}_n$ correspond à n dépliages de la fonction $\texttt{length} = \texttt{length}_0$ ordinaire, qui calcule la longueur d'une liste L. Sans donner le détail d'un modèle d'exécution ni d'un modèle de performance, on peut supposer qu'il lui correspond un coût de la forme

$$\mu(\texttt{length}_n\,(L)) = |L|\,c + \left\lfloor \frac{|L|}{n+1} \right\rfloor c_{\texttt{CALL}}$$

où $|L|$ est la longueur effective de la liste L, c est le coût d'exécution d'un `case`, et $c_{\texttt{CALL}}$ est le coût spécifique de l'appel fonctionnel « $\texttt{length}_n\,(L_{n+1})$ ». Pour tout L, on a d'une part $\mu(\texttt{length}_n\,(L)) \succcurlyeq \mu(\texttt{length}_{n+1}\,(L))$ et d'autre part $\mu(\texttt{length}_n\,(L)) \succcurlyeq \mu(\texttt{length}_{n+1}\,(L)) + c_{\texttt{CALL}}$ ssi $|L| \geqslant (n+1)(n+2)$. Par conséquent, pour tout $n \in \mathbb{N}$, $\texttt{length}_{n+1} \prec_{\text{abs}} \texttt{length}_n$. La suite de programmes $(\texttt{length}_n)_{n\in\mathbb{N}}$ n'est pas *bien fondée* pour $\prec_{\text{abs}}$ (cf. §N.9). Cela ne suffit pas en soi à démontrer que la fonction `length` n'admet pas d'évaluation partielle minimale. Cependant, le même procédé appliqué à une hypothétique évaluation partielle minimale `lengthMin`, nécessairement récursive, trouve une fonction $\texttt{lengthMin}_1 \prec_{\text{abs}} \texttt{lengthMin}$, ce qui contredit la minimalité de `lengthMin`. Par conséquent, il n'y a pas non plus d'évaluation partielle optimale.

Parce que les coûts moyens sont stables et discriminants, on a $\prec_{\text{abs}} \subset \prec_{\text{moy}}$ (prop. 3.3). Les comparaisons $\preccurlyeq_{\text{moy-fini}}$, $\preccurlyeq_{\text{moy-pfini}}$ et $\preccurlyeq_{\text{moy-lim}}$ vérifient également $\prec_{\text{abs}} \subset \prec_{\text{moy-fini}}, \prec_{\text{moy-pfini}}, \prec_{\text{moy-lim}}$ car leur construction préserve la stabilité et la discrimination (tabl. 3.4, 3.6.5). Par conséquent, la suite de programmes $(\texttt{length}_n)_{n\in\mathbb{N}}$ n'est également pas bien fondée pour $\prec_{\text{moy-fini}}$, $\prec_{\text{moy-pfini}}$ et $\prec_{\text{moy-lim}}$. On n'a donc généralement pas non plus d'évaluation partielle minimale ou optimale pour le coût moyen sur un domaine infini.

Spécifications algébriques.

Pour être optimale ou minimale, une évaluation partielle, et en particulier une spécialisation, doit faire « un usage maximum des informations présentes dans les données statiques ». C'est d'ailleurs parfois comme cela que l'on décrit (ou que l'on a décrit) informellement ce qu'est l'évaluation partielle.

Dans le cas où le modèle d'exécution est donné dans un formalisme algébrique (logique équationnelle, algèbre initiale, système de réécriture), il est nécessaire pour cela de rajouter des équations à la spécification jusqu'à ce qu'elle soit, le cas échéant, ω-complète. Or il existe des spécifications qui n'admettent pas d'enrichissement ω-complet ; il est alors impossible d'exploiter toutes ces informations, et donc matériellement de réaliser un évaluateur partiel optimal. Nous renvoyons le lecteur à [Hee86] pour plus de détails sur ce point.

4.5.2 Conditions d'existence.

Le problème fondamental de l'évaluation partielle optimale ou minimale est que l'on doit comparer un programme à *tous* ceux qui lui sont équivalents, programmes qui en pratique sont en nombre infini et que l'on ne peut même pas dénombrer du fait de l'indécidabilité de l'équivalence.

Évaluation partielle optimale.

En fait, il n'existe guère qu'une méthode pour assurer l'existence d'une évaluation partielle optimale d'un programme p. Elle procède en deux étapes : il faut tout d'abord *minorer* le coût d'exécution dans P, c'est-à-dire trouver une fonction

$$f \in \mathcal{C}^{\mathrm{Dom}(p)} \quad \text{telle que} \quad \text{pour tout } p' \in P \quad f \preccurlyeq \mu(p')$$

En pratique, il faut pour cela examiner toutes les constructions sémantiques qui sont *indispensables* pour qu'un programme p' soit équivalent à p. Pour ce type de minoration, il est nécessaire de disposer d'une propriété d'additivité. La seconde étape consiste à *exhiber* un programme q qui *atteint* le coût f. On a alors :

$$\text{si} \quad \mu(q) = f \quad \text{alors} \quad q \in \mathit{Evap}^{\mathrm{optF}}_{P,\preccurlyeq}(p)$$

Un tel programme est une évaluation partielle optimale. Considérons par exemple le programme `not` suivant :

```
fun not(X) = if X then false else true
```

Pour programmer la fonction mathématique $not = Bool(\mathtt{not}) : \mathrm{BoolVal} \rightarrow \mathrm{BoolVal}$, il est indispensable de tester la valeur de l'argument car le résultat en dépend. Il faut donc au minimum une construction `if` dont la condition est une expression qui contient la variable `X`. Ensuite, selon le résultat du test, il faut retourner soit $true$, soit $false$; puisque la variable `X` contient la valeur opposée, il faut nécessairement faire apparaître à un moment donné une construction `false` ou `true`. Le coût d'exécution de tout programme équivalent à not est donc uniformément minoré par $c_{\mathtt{if}} + c_{var} + c_{bool}$. Or, le programme `not` lui-même atteint ce minorant. C'est donc une évaluation partielle optimale. Il s'agit en fait là du raisonnement que nous avons déjà tenu pour justifier les évaluations partielles optimales de la section §4.4.

La proposition 4.7, donnée à la section suivante (§4.6), indique comment obtenir une évaluation partielle optimale à partir de résultats obtenus pour un autre langage. Un cas particulier (prop. 4.8) donne un cadre de construction explicite d'une évaluation partielle optimale.

Évaluation partielle minimale.

Dès qu'interviennent des dépendences sur plusieurs arguments (ou sur un argument structuré), apparaît aussi un choix sur l'ordre des tests[2]. La minoration se traduit alors non plus par une fonction unique mais par un ensemble de fonctions

$$F \subset \mathcal{C}^{\mathrm{Dom}(p)} \quad \text{tel que} \quad \text{pour tout } p' \in P \quad \text{il existe } f \in F \quad f \preccurlyeq \mu(p')$$

Comme précédemment, on examine toutes les constructions sémantiques indispensables au fur et à mesure que l'on accroît l'*information* connue sur les données d'entrée. Une décision doit intervenir au moment de

[2] Nous considérons ici des *machines séquentielles* et, par là-même, des *algorithmes séquentiels* (voir aussi [Ber81, BC82]).

choisir *où* gagner de l'information sur ces données afin de mieux cerner le résultat. S'ajoute ensuite une étape intermédiaire qui consiste à déterminer l'ensemble $F_{\min} = \mathrm{Emin}_{\preccurlyeq}(F)$ des éléments minimaux de F. On sait que cet ensemble est non-vide lorsque F est bien fondé ; c'est le cas en particulier lorsque F est fini. Pour conclure, il faut exhiber un programme q de P qui atteigne un quelconque des coûts de $F_{\min}$.

$$\text{si} \quad \mu(q) \in F_{\min} \quad \text{alors} \quad q \in Evap^{\min}_{P,\preccurlyeq}(p)$$

En effet, si $\mu(q) \in F_{\min}$ alors pour tout programme $p' \in P$, il existe un $f \in F$ tel que $f \preccurlyeq \mu(p')$ et $f \not\prec \mu(q)$. Supposer $\mu(p') \prec \mu(q)$ conduit à une contradiction car alors $f \prec \mu(q)$. Par conséquent, $\mu(p') \not\prec \mu(q)$ et q est bien une évaluation partielle minimale.

À titre d'exemple, appliquons cette méthode sur la fonction mathématique $nand(X, Y)$. Puisque le résultat dépend de la valeur de chacun des arguments, il y a au minimum un test sur chacun d'eux. Si l'on commence par X, il est un cas où il n'est pas indispensable de connaître d'information sur Y. La situation est symétrique si l'on débute par Y. Dans chaque cas, il est toujours plus avantageux pour fournir la valeur retournée de la construire explicitement plutôt que d'utiliser une variable ou toute autre construction. Par conséquent, on a $F = \{f_1, f_2\}$ avec :

$$\begin{array}{rcl} f_1(X,Y) &=& \text{si } (X = true) \text{ alors } 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \text{ sinon } c_{\mathtt{if}} + c_{var} + c_{bool} \\ f_2(X,Y) &=& \text{si } (Y = true) \text{ alors } 2\,c_{\mathtt{if}} + 2\,c_{var} + c_{bool} \text{ sinon } c_{\mathtt{if}} + c_{var} + c_{bool} \end{array}$$

Puisque F est fini, on sait que $F_{\min} = \mathrm{Emin}(F)$ est non-vide ; en l'occurrence, $F_{\min} = F$ car $f_1 \not\lesseqgtr_{\text{abs}} f_2$. Or, les programmes `nand1` et `nand2` atteignent respectivement des fonctions f_1 et f_2 de $F_{\min}$. Sans connaître les autres programmes `nand`i, on peut alors affirmer que `nand1` et `nand2` sont des évaluations partielles minimales de tout programme de sémantique $nand$.

Existence pour les domaines finis.

S'il n'est pas possible d'établir un théorème *formel* général d'inexistence d'une évaluation partielle optimale et minimale pour les programmes de domaine *infini*, parce que le dépliage d'une construction itérative reste trop spécifique au langage considéré, il est en revanche possible de garantir l'existence d'une évaluation partielle minimale pour les coûts absolus, moyen et maximum, à condition que le programme soit de domaine *fini*.

Proposition 4.5. (Condition d'existence d'une évaluation partielle minimale)
Soient $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ un langage, $(\mathcal{C}, +, ., \preccurlyeq)$ un domaine de coût algébrique additif, $(c_i)_{i \in I}$ une famille de coûts positifs au plus dénombrable et bien fondée, et μ une mesure de performance de L à valeur dans $\sum_{i \in (I)} \mathbb{N} c_i$. Alors les modèles de coût absolu, moyen et maximum construits sur $\preccurlyeq$ vérifient la proposition suivante (nous notons $\preccurlyeq_D \in \{\preccurlyeq_{\text{abs},D}, \preccurlyeq_{\text{moy},D}, \preccurlyeq_{\text{max},D}\}$) :

- $\forall p \in \mathcal{P} \quad \forall P \subset \mathcal{P}$ tq $Equiv_{\text{str}}(p) \subset P \subset Equiv_{\text{par}}(p)$, si $Dom(p)$ est fini alors $Evap^{\min}_{P,\preccurlyeq_{Dom(p)}}(p) \neq \varnothing$

Cette proposition s'applique en particulier si la famille $(c_i)_{i \in I}$ est finie car elle est alors nécessairement bien fondée.

Il n'y a pas de proposition similaire pour l'évaluation partielle optimale. Qu'il n'y ait pas non plus de résultat similaire pour une comparaison de coût quelconque n'est pas très génant en pratique. En effet, puisque le domaine est fini, beaucoup de comparaisons coïncident (cf. §3.6.6) avec les coûts absolu, moyen et maximum.

Notons enfin que la démonstration qui suit est non-constructive ; elle ne donne pas de moyen explicite de calculer une évaluation partielle minimale.

Démonstration. Pour simplifier la démonstration, nous supposons les coûts $(c_i)_{i\in I}$ indicés par les entiers naturels ; nous supposons plus précisément que I est un intervalle de $\mathbb{N}$ contenant 0. Soit $(\kappa_n)_{n\in\mathbb{N}}$ une suite de coûts de $\sum_{i\in(I)} \mathbb{N}c_i$ strictement décroissante : pour tout $n \in \mathbb{N}$, $\kappa_n \succ \kappa_{n+1}$.

Puisque $\kappa_n \in \sum_{i\in(I)} \mathbb{N}c_i$, il existe une famille presque nulle d'entiers naturels $(\alpha_{n,i})_{i\in I}$ telle que $\kappa_n = \sum_{i\in I} \alpha_{n,i}\, c_i$. Soient j_n le plus grand indice des coûts élémentaires qui interviennent dans $\kappa_0, \ldots, \kappa_n$, et s_n la séquence des coefficients des coûts élémentaires de κ_n pour les indices inférieurs à j_n :

- $j_n = \max(\{j \in I \mid \exists\, l \leqslant n \;\; \alpha_{l,j} \neq 0\})$
- $s_n = \langle \alpha_{n,i} \mid 0 \leqslant i \leqslant j_n \rangle$

Notons que $(s_n)_{n\in\mathbb{N}}$ est une suite de séquences de longueurs croissantes et que $\kappa_n = \sum_{0\leqslant i\leqslant j_n} \alpha_{n,i}\, c_i$.

D'autre part, la relation $\leqslant$ sur $\mathbb{N}$ est un *bon ordre* (cf. §N.10) :

- $\forall (x_n)_{n\in N} \in \mathbb{N}^{\mathbb{N}} \;\; \exists\, i < j \in \mathbb{N} \;\; x_i \leqslant x_j$

On note $\trianglelefteq$ le prolongement de $\leqslant$ aux séquences finies :

- $\forall n \leqslant m \in \mathbb{N} \;\; \forall x_1, \ldots, x_n, y_1, \ldots, y_m \in \mathbb{N} \;\; (x_1, \ldots, x_n) \trianglelefteq (y_1, \ldots, y_m)$ ssi $\forall i \in [n] \;\; x_i \leqslant y_i$

C'est un cas particulier de *plongement dans les séquences* (embedding) (cf. §N.10). La relation $\trianglelefteq$ définit un ordre sur les séquences finies d'entiers naturels. Grâce au lemme de Higman (cf. §N.10), on sait que $\trianglelefteq$ est alors aussi un bel ordre.

Par conséquent, il existe $n < m \in \mathbb{N}$ tels que $s_n \trianglelefteq s_m$, c'est-à-dire $\forall\, 0 \leqslant i \leqslant j_n \;\; \alpha_{n,i} \leqslant \alpha_{m,i}$. Donc, $\kappa_n = \sum_{0\leqslant i\leqslant j_n} \alpha_{n,i} c_i \preccurlyeq \sum_{0\leqslant i\leqslant j_n} \alpha_{m,i} c_i \preccurlyeq \sum_{0\leqslant i\leqslant j_m} \alpha_{m,i} c_i = \kappa_m$ par additivité, ce qui contredit l'hypothèse sur $(\kappa_n)_{n\in N}$. Il n'existe donc pas de suite infinie de coûts de $\sum_{i\in(I)} \mathbb{N}c_i$ strictement décroissante.

Considérons à présent un support D fini. Les mesures de performance explicites $\mu_{\mathrm{moy},D}$ et $\mu_{\mathrm{max},D}$ (déf. 3.24, 3.29) sont à valeur dans $\sum_{i\in(I)} \mathbb{N}c_i$. Donc $(\mathcal{P}_D, \preccurlyeq_{\mathrm{moy},D,\mu})$ et $(\mathcal{P}_D, \preccurlyeq_{\mathrm{max},D,\mu})$ sont également bien fondés.

D'autre part, on sait que le produit fini d'ensembles bien fondés est bien fondé pour la relation produit. Puisque D est un ensemble fini, $(\sum_{i\in(I)} \mathbb{N}c_i)^{|D|}$ est bien fondé pour $\preccurlyeq^{|D|}$, et donc $(\sum_{i\in(I)} \mathbb{N}c_i)^D$ bien fondé pour $\preccurlyeq_{\mathrm{abs},D}$. En particulier, $(\mathcal{P}_D, \preccurlyeq_{\mathrm{abs},D,\mu})$ est également bien fondé.

En résumé, $(\mathcal{P}_D, \preccurlyeq_{D,\mu})$ est bien fondé pour $\preccurlyeq_D \in \{\preccurlyeq_{\mathrm{abs},D}, \preccurlyeq_{\mathrm{moy},D}, \preccurlyeq_{\mathrm{max},D}\}$. Donc pour tout programme $p \in P$ de domaine fini et tout $P \subset \mathcal{P}$ tel que $\mathit{Equiv}_{\mathrm{str}}(p) \subset P \subset \mathit{Equiv}_{\mathrm{par}}(p)$, il n'existe également pas de suite de programmes de P strictement décroissante pour $\prec_{\mathit{Dom}(p),\mu}$. Par conséquent, le processus qui consiste à partir de $p_0 = p$ et à choisir un $p_{n+1} \prec_{\mathit{Dom}(p)} p_n$ dans P pour tout $n \in \mathbb{N}$ s'arrête donc après un nombre fini m d'étapes. On a alors $p_m \in \mathit{Evap}^{\min}_{P,\preccurlyeq_{\mathit{Dom}(p)}}(p)$. $\square$

Les conditions de la proposition 4.5 étaient remplies par le modèle abstrait de performance dynamique de BOOL (cf. §3.2.5), additif par hypothèse, dont les coût élémentaires vérifient $0 \prec c_{\mathit{bool}} \prec c_{\mathit{var}} \preccurlyeq c_{\mathtt{if}} \prec c_{\mathit{fun}_n} \prec c_{\mathit{fun}_m}$ pour $0 \leqslant n < m$ et sont donc bien fondés. A posteriori, il n'est pas donc étonnant que nous ayons effectivement trouvé une évaluation partielle minimale pour les coûts absolus, moyens, et maximum (cf. §4.4).

4.5.3 Méthodes d'obtention.

Jusqu'à présent, nous n'avons pas indiqué comment *fabriquer* une évaluation partielle optimale (resp. minimale), mais simplement comment être sûr qu'un programme *donné* est ou non optimal (resp. minimal) (cf. §4.5.2). Nous envisageons ici diverses méthodes pour cela. La section suivante (§4.6) propose quant à elle un moyen de transporter un programme optimal (resp. minimal) d'un langage dans un autre.

Échec de la construction par composition.

La composabilité (cf. §3.5.5) d'une comparaison de coût permet de fabriquer des évaluations partielles de manière « compositionnelle ». Soient par exemple les programmes SML suivants

```
fun p1(x) = append([1,2],x)
fun p2(x) = 1::2::x
```

ainsi qu'une mesure de coût « plausible » vérifiant `p2` $\preccurlyeq$ `p1`. Formons les programmes suivants :

```
fun q1(x) = [p1(x),p1(x)]
fun q2(x) = [p2(x),p2(x)]
```

Si $\preccurlyeq$ est faiblement composable à gauche, ou de manière équivalente faiblement additive (prop. 3.2.9), on a alors la relation `q2` $\preccurlyeq$ `q1`. Plus précisément, la faible composabilité à gauche (déf. 3.40) signifie :

$$\forall p \in \mathcal{P} \quad \forall p' \in Evap_{\preccurlyeq}(p) \quad \forall q \in \mathcal{P}_{Im(p)} \quad q \circ p' \in Evap_{\preccurlyeq}(q \circ p)$$

Cette propriété n'est pas vraie pour l'évaluation partielle optimale. Si l'on a « vraisemblablement » `p2` $\in Evap^{\mathrm{opt}}($`p1`$)$, on n'a pas nécessairement l'appartenance de `q2` à $Evap^{\mathrm{opt}}($`q1`$)$. D'ailleurs, le programme `q2'` suivant est strictement meilleur que `q2`.

```
fun q2'(x) = let val y = p2(x) in [y,y] end
```

Cela a des conséquences sur les méthodes à mettre en œuvre pour construire ou prouver la validité des évaluateurs partiels. Comme le suggère la composabilité à gauche, l'appartenance à l'évaluation partielle peut faire appel à la *récurrence* (structurelle). En revanche, l'évaluation partielle optimale ne peut y avoir recours ; elle nécessite des preuves *globales* et « *directes* », très spécifiques. Cela se traduit également sur la complexité d'un éventuel algorithme d'évaluation partielle optimale, qui doit souvent examiner la quasi totalité des programmes possibles. Par exemple, la recherche d'une compilation optimale des motifs de SML est NP-complète [App92]. Cela nous amène à la méthode par énumération.

Méthode par énumération.

Dans le cas exceptionnel où l'on dispose d'un moyen de décider l'équivalence de deux programmes, il est envisageable d'énumérer systématiquement des programmes candidats p' et de tester d'une part leur équivalence à p et d'autre part leur adéquation au critère de performance $p' \preccurlyeq p$.

Cette recherche peut être *exhaustive* du point de vu de l'optimum si l'on peut assurer à un certain point que tous les programmes p' restant à examiner ont un coût supérieur à p (resp. ne sont pas de coût inférieur à p). Sans cette garantie, on est peu avancé car non seulement il est possible d'énumérer une infinité de programmes qui ne sont pas équivalents à p, mais même lorsqu'ils le sont, ils peuvent être de coût supérieur à celui de p ; dans ce cas, le programme p reste le seul meilleur représentant connu de $L(p)$.

Cette recherche même a un coût évidemment explosif, malgré les heuristiques employées pour essayer de la contenir. Elle est néanmoins justifiée dans de rares cas finis[3].

[3] Il existe des *superoptimiseurs* [Mas87] conçus pour faire la recherche (quasi-)exhaustive de *meilleures* séquences d'instructions machine. Ils prennent en paramètre la description d'une machine physique et un petit fragment de code. Leur but est de réaliser des librairies de fonctions optimales, destinées à être substituées « in-line » directement dans le code généré d'un compilateur. Il s'agit par

Transformations monotones.

En pratique, les évaluations partielles sont obtenues par applications successives de transformations correctes. On peut garantir une optimisation (resp. une optimisation stricte) si l'on emploie des transformations monotones (resp. strictement monotones) par rapport au critère de performance.

Même si l'on sait qu'il existe bien un optimum, l'itération de ce procédé de l'atteint pas nécéssairement. Ce peut être pour l'une des raisons suivantes :

- Les transformations deviennent stationnaires du point de vue du coût (si la monotonie n'est pas stricte).
- Les améliorations sont infinitésimales et tendent vers un optimum sans jamais l'atteindre. C'est contre cela qu'existe la condition de bonne fondation dans la proposition 4.5.
- Les transformations améliorent indéfiniment, au moins d'un coût constant fixé. Ce n'est pas incompatible avec l'existence supposée d'un minimum car on opère sur des objets fonctionnels : on peut par exemple optimiser un programme *successivement* sur chacune des données de son domaine sans atteindre le programe qui est optimal sur *toutes* les données.
- On s'est échoué dans un minimum local ; il n'y a plus de transformation disponible qui fasse décroître le coût.
- On est interrompu prématurément par un critère d'arrêt (cf. §6.2) à cause du danger de non-terminaison.

Notons par ailleurs que, même lorsqu'il existe un optimum unique, il n'y a pas nécessairement confluence des transformations.

4.6 Modèle intermédiaire.

Pour étudier les propriétés des évaluations partielles (notamment optimales et minimales), et pour les calculer de manière effective, il est parfois plus commode d'utiliser un langage intermédiaire qui « respecte » en quelque sorte la sémantique et la mesure de performance d'origine, mais qui simplifie (grâce à plus d'homogénéité, plus de régularité) les représentations des programmes et leurs classes d'équivalence de coût.

4.6.1 Motivation.

Pour motiver cette approche, nous définissons un nouveau langage, ZML, constitué uniquement de constructeurs de types de données (il n'y a pas de conditionnelles ni d'appels fonctionnels) et de déclarations de variables.

exemple de fonctions comme `abs`, `max`, la multiplication par des constantes, etc. En l'occurrence, le calcul du critère de performance (comparaison du nombre de temps de cycles) est immédiat, et deux ou trois tests d'équivalence sur des données aléatoires suffisent à rejeter la plupart des propositions erronées. Pour les cas restants, il faut démontrer l'équivalence avec exactitude. La terminaison de la recherche exhaustive est assurée par la monotonie du critère. Du fait de l'explosion combinatoire, la taille du code superoptimisable en pratique est de l'ordre de 5 à 6 instructions. Des résultats partiels peuvent être obtenus en déplaçant une *fenêtre* (peephole) d'optimisation de taille réduite sur une séquence de code plus longue. On cite le cas d'un calcul de deux semaines pour obtenir un gain substantiel dans une séquence de code *critique* comportant une vingtaine d'instructions.

Le superoptimiseur de GNU [GK92] est crédité d'une performance $\mathcal{O}(m^n n^{2n})$ où m est le nombre d'instructions possibles, et n est la plus courte séquence d'instructions de la fonction à réaliser. En pratique toutefois, les valeurs immédiates (les constantes) ne sont pas prises en compte et c'est surtout sur les mouvements de registres qu'il donne toute sa puissance. De plus, son test de correction n'est pas totalement sûr ; il est limité à des heuristiques.

Les « programmes » ZML sont en quelque sorte un noyau d'expressions constantes de SML, le « degré zéro » de ML.

Ils sont évidemment peu intéressants intrinsèquement ; leur domaine de définition est un singleton et ils n'ont qu'une seule exécution, de résultat constant. Cette simplicité n'est pas vaine car ils représentent en fait les expressions *terminales* d'un programme SML, dont l'évaluation constitue la valeur de retour de tout appel fonctionnel. Le problème de l'évaluation partielle revient à savoir comment programmer au mieux le calcul de ces constantes.

Syntaxe. En plus des constructeurs de types de données et des lieurs de variables, nous considérons également des délimiteurs de portée lexicale (constructions `let` et `local`). La syntaxe complète de ZML est donnée à la figure 4.1. Pour simplifier, nous ne distinguons pas, comme en SML, la construction d'un 0-uplet et la construction nullaire, et nous notons par exemple `NIL` plutôt que `NIL()`. Nous ne nous préoccupons pas ici de la déclaration des types de données, que nous notons en majuscules pour les distinguer des variables notées en minuscules. Nous supposons de plus qu'une construction $valbinds = valbind_1$ `and` $\ldots$ `and` $valbind_n$ ne lie pas la même variable plusieurs fois.

exp	::=	var	`x, y, z...`V
		$con\langle$`(`exp_1`, ... ,`exp_n`)`$\rangle$	`NIL, F(G(A),B), CONS(y,NIL)`
		`let` $decs$ `in` exp `end`	`let val x = ZERO in CONS(x,NIL) end`
$valbind$	::=	var `=` exp	`x = F(G(A),B)`
$valbinds$	::=	$valbind$ $\langle$`and` $valbinds\rangle$	`x = y and y = x`
dec	::=	`val` $valbinds$	`val x = SUCC(SUCC(ZERO))`
		`local` $decs_1$ `in` $decs_2$ `end`	`local val x = ZERO in val y = SUCC(x) end`
$decs$	::=		*déclaration vide*
		dec $\langle$`;`$\rangle$ $decs$	`val x = ZERO; val y = CONS(x,NIL)`
$prog$	::=	$decs$`;`	`val x = SUCC(SUCC(ZERO));`

Figure 4.1 : Syntaxe de ZML

Objets sémantiques. Les objets sémantiques employés dans la spécification de ZML en sémantique naturelle sont donnés à la figure 4.2, accompagnés de quelques exemples. Le résultat de l'évaluation d'un programme ZML est un environnement de valeurs associant un type de données à des noms de variable.

var	$\in$	Var	$x,\ y,\ z\ldots$
con	$\in$	Con	$\mathit{ZERO},\ \mathit{SUCC},\ \mathit{NIL},\ \mathit{CONS}\ldots$
v	$\in$	$\mathrm{Val} = \bigcup_{k\geqslant 0}(\mathrm{Con}\times\mathrm{Val}^k)$	$\mathit{CONS}(\mathit{SUCC}(\mathit{ZERO}),\mathit{NIL})$
E	$\in$	$\mathrm{Env} = \mathrm{Var} \overset{\text{fin}}{\to} \mathrm{Val}$	$\{x \mapsto \mathit{NIL}, y \mapsto \mathit{SUCC}(\mathit{ZERO})\}$

Figure 4.2 : Objets sémantiques de ZML

Sémantique. La figure 4.3 spécifie en sémantique naturelle les preuves du jugement $E_0 \vdash prog \Rightarrow E$, qui exprime qu'un programme $prog$ produit un environnement E dans le contexte d'un environnement de départ E_0. Nous nous plaçons dans le cas où cet environnement E_0 initial est toujours vide[4] :

$$Zml(prog) = E \quad \text{ssi} \quad \varnothing \vdash prog \Rightarrow E$$

Par exemple, le programme « `local val x=F(A) in val y=G(x,H(x)) end` » s'évalue dans l'environnement vide en $E = \{y \mapsto G(F(A), H(F(A)))\}$. À quelques adaptations mineures, ces régles sont identiques à celles de la définition formelle de SML [MTH90].

$$\frac{}{E \vdash var \Rightarrow E(var)} : c_{var} \qquad (\rho_{var})$$

$$\frac{\langle E \vdash exp_1 \Rightarrow v_1 \quad \ldots \quad E \vdash exp_n \Rightarrow v_n \rangle}{E \vdash con\langle \texttt{(}exp_1\texttt{,} \ldots \texttt{,}exp_n\texttt{)}\rangle \Rightarrow con\langle (v_1\texttt{,} \ldots \texttt{,} v_n)\rangle} : c_{con_n} \qquad (\rho_{con_n})$$

$$\frac{E \vdash dec \Rightarrow E' \qquad E + E' \vdash exp \Rightarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v}$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash var\ \texttt{=}\ exp \Rightarrow \{var \mapsto v\}} : c_{valbind} \qquad (\rho_{valbind}))$$

$$\frac{E \vdash valbind \Rightarrow E' \qquad \langle E \vdash valbinds \Rightarrow E'' \rangle}{E \vdash valbind\ \langle \texttt{and}\ valbinds \rangle \Rightarrow E'\ \langle +E'' \rangle}$$

$$\frac{E \vdash valbind \Rightarrow E'}{E \vdash \texttt{val}\ valbind \Rightarrow E'}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow E_2}$$

$$\frac{}{E \vdash \quad \Rightarrow \{\}}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash dec_1\texttt{;}\ dec_2 \Rightarrow E_1 + E_2}$$

Figure 4.3 : Sémantique dynamique de ZML

Modèle de performance. Les règles de la figure 4.3 sont en outre annotées par des coûts élémentaires qui définissent un modèle d'exécution de ZML (cf. §3.2.5). Nous nous plaçons dans le cadre d'un domaine de coût abstrait $(\mathcal{C}, +, ., \preccurlyeq)$ additif, réflexif et total (cf. §3.2.5, domaine de coût abstrait). La mesure de performance

[4] Voir aussi la note de bas de page numéro 7.

μ que définissent ces coûts élémentaires est justifiée par l'implémentation informelle suivante (cf. §3.2.5, implémentation informelle) :

- Le coût d'accès à la valeur d'une variable (règle ρ_{var}) est un coût constant c_{var}. Cette hypothèse n'est pas vérifiée avec les modèles de compilation SECD ou CAM qui font de l'environnement local une liste ou un arbre binaire, plutôt qu'un vecteur ; le temps d'accès est alors linéaire ou logarithmique. En outre, les variables ne sont pas toujours intégrées dans un environnement explicite (pile, fermeture...), mais parfois compilées comme des registres machine, d'accès plus rapide et sujets à de multiples optimisations (notamment avec les architectures RISC).

- Un coût c_{con_n} est attribué (règle ρ_{con_n}) à la construction $con(exp_1, \ldots, exp_n)$. Il comprend l'allocation et l'initialisation du vecteur ou de la liste nécessaire. Le coût d'une construction nullaire est c_{con_0}.

- De même que c_{var}, le coût $c_{valbind}$ d'une liaison de valeur (règle $\rho_{valbind}$) varie beaucoup en fonction des optimisations de compilation. La valeur peut momentanément être stockée dans une pile, une fermeture, un registre. En cas d'appel fonctionnel, le type de sauvegarde diffère aussi suivant qu'elle est de la responsabilité de la fonction appelante ou appelée [ASU86].

  Pour simplifier, nous donnons un prix constant à l'opération comprennant l'allocation de la cellule destinée à contenir v, son initialisation, et également sa libération à la restauration de l'environnement. Ce dernier point a pour but d'annuler le prix des constructions `let` et `local` : le coût global du programme est alors surévalué d'une valeur constante correspondant à N libérations où N est le nombre de liaisons qui apparaissent dans l'environnement E final. Nous travaillons modulo cette translation constante pour un environnement résultat E donné.

- Nous considérons nul ou négligeable le coût des autres règles, soit parce qu'il a été incorporé à d'autres coûts, comme c'est le cas pour les constructions `let` et `local`, soit parce qu'il correspond simplement à l'enchaînement des instructions compilées.

Par exemple, le programme « `local val x=F(A) in val y=G(x,H(x)) end` » a pour coût d'exécution $2\,c_{valbind} + c_{con_0} + 2\,c_{con_1} + c_{con_2} + 2\,c_{var}$.

Optimisation. L'optimisation dans ZML se résume à un usage pertinent des constructions `let` et `local`. L'essence du `let`, au moins sur le plan opérationnel, est le partage. Considérons les deux expressions équivalentes $exp_1 =$ `G(F(A),H(F(A)))` et $exp_2 =$ `let x = F(A) in G(x,H(x)) end`. La première a un coût d'exécution $c_{con_2} + 3\,c_{con_1} + 2\,c_{con_0}$ et la seconde $c_{con_2} + 2\,c_{con_1} + c_{con_0} + 2\,c_{var} + c_{valbind}$. Il faut préférer exp_1 ou exp_2 suivant que $c_{con_1} + c_{con_0} \preccurlyeq c_{valbind} + 2\,c_{var}$ ou non[5].

L'évaluation partielle optimale (ou minimale car $\preccurlyeq$ est total, prop. 4.4.6) existe toujours (prop. 4.5), et repose sur le même type de choix. Reprenons la méthode proposée à la section §4.5.2. Soit $prog$ un programme de Prog qui s'évalue en $Zml(prog) = E$ et notons $\mathrm{nb}(v, E)$ le nombre d'occurrences d'une valeur v parmi les sous-termes de E. Pour produire $v = con_n(v_1, \ldots, v_n)$ en supposant $v_1, \ldots, v_n$ disponibles, il y a deux moyens possibles : ou bien construire v en appliquant le constructeur con à $v_1, \ldots, v_n$, ou bien aller chercher v dans l'environnement courant en accédant à une variable, mais il faut pour cela d'une part que la valeur ait déjà été créée (au moins une fois), d'autre part qu'elle ait été auparavant liée grâce à une construction $valbind$.

[5] Indépendamment du modèle d'exécution, nous avons pu vérifier par exemple que le compilateur « STANDARD ML of New Jersey » version 0.93 des Bell Labs ne créait jamais de partage physique implicite des termes. La différence de temps d'exécution sur SUN4 entre « `G(F(A),H(F(A)))` » et « `let val x=F(A) in G(x,H(x)) end` » est de l'ordre de 15% en faveur du partage.

Le coût d'exécution de tout programme qui s'évalue en E est donc minoré par (la somme est presque nulle) :

$$\mu_{\min}(E) = \sum_{v \in \mathrm{Val}} \min_{\substack{i \geqslant 1 \\ i+j = \mathrm{nb}(v,E) \\ k = \text{si } j \geqslant 1 \text{ alors } 1 \text{ sinon } 0}} (i\, c_{con_n} + j\, c_{var} + k\, c_{valbind})$$

Arrivé à ce point, il est cependant malaisé de mettre en œuvre directement la suite de la méthode de la section §4.5.2 car choisir s'il faut construire de toute pièce une nouvelle valeur, la sauvegarder pour une réutilisation éventuelle, ou la récuper dans l'environnement courant, est compliqué par la possibilité d'encapsuler les liaisons au moyen des constructions `let` et `local`. La structure des programmes, trop concrète, est un obstacle au raisonnement sur l'optimisation.

4.6.2 Théorie.

C'est pourquoi nous examinons dans quelles circonstances on peut faire « migrer » le problème vers un autre langage, en pratique plus simple, plus régulier. On désire savoir si cette mise en correspondance transporte ou non des résultats d'évaluation partielle, éventuellement optimale. À la section suivante, nous mettrons en œuvre ce type de démarche afin de construire explicitement un évaluateur partiel optimal fort de ZML (cf. §4.6.3). Nous employons dans notre approche une variante[6] de la notion de connexion de Galois [MSS85, HH85].

Définition 4.5. (Connexion de Galois contractante)
Soient $(E, \preccurlyeq)$ et $(E', \preccurlyeq')$ deux ensembles préordonnés, et $f : E' \to E, f' : E \to E'$. La paire de fonctions (f', f) forme une *connexion de Galois contractante entre* $(E, \preccurlyeq)$ *et* $(E', \preccurlyeq')$ ssi :

- $\forall x, y \in E \quad x \preccurlyeq y \Rightarrow f'(x) \preccurlyeq' f'(y)$ et $\forall x', y' \in E' \quad x' \preccurlyeq' y' \Rightarrow f(x') \preccurlyeq f(y')$
- $\forall x \in E \quad f(f'(x)) \preccurlyeq x$ et $\forall x' \in E' \quad f'(f(x')) \preccurlyeq' x'$

Autrement dit, les fonctions f et f' sont croissantes et leurs composées inférieures à l'identité.

Intuitivement, les ensembles $(E, \preccurlyeq)$ et $(E', \preccurlyeq')$ vont représenter les programmes de deux langages différents, chacun muni d'une relation de coût, et les fonctions f et f', des traductions (transformations) d'un langage dans l'autre. Les conditions requises signifient d'une part que les comparaisons de programmes qui sont vraies pour un langage le sont également après traduction dans l'autre, et d'autre part que l'on ne perd pas de performance en appliquant deux fois les transformations et en revenant ainsi au langage de départ.

Examinons comment les connexions de Galois contractantes se comportent sur les ensembles optimaux que nous avons définis à la section §4.2.

Proposition 4.6. (Connexion de Galois contractante et optimalité)
Soient (f', f) une connexion de Galois contractante entre $(E, \preccurlyeq)$ et $(E', \preccurlyeq')$, et A une partie de E.

- $f'(\mathrm{Emin}_{\preccurlyeq}(A)) \subset \mathrm{Emin}_{\preccurlyeq'}(f'(A))$
- $f'(\mathrm{Emin}_{\preccurlyeq|A}(E)) \subset \mathrm{Emin}_{\preccurlyeq'|f'(A)}(f'(E))$
- $f'(\mathrm{Min}_{\preccurlyeq}(A)) \subset \mathrm{Min}_{\preccurlyeq'}(f'(A))$
- $f'(\mathrm{Min}_{\preccurlyeq|A}(E)) \subset \mathrm{Min}_{\preccurlyeq'|f'(A)}(f'(E))$

[6] La définition ordinaire dit que les fonctions f et f' sont décroissantes et que leur composition est supérieure à l'identité. Si on l'applique aux relations réciproques $\succcurlyeq$ et $\succcurlyeq'$, on obtient notre définition à la différence près que f et f' sont encore décroissantes et non croissantes. Avec ce type de condition, on n'obtient pas les résultats que nous donnons par la suite. C'est pourquoi nous avons dû introduire la variante suivante.

La définition de connexion de Galois contractante étant symétrique, on a les mêmes résultats pour l'application de f sur les ensembles optimaux de E'.

Avant de reprendre les définitions des évaluations partielles optimales, nous précisions la notion de correspondance dont nous avons parlé en introduction.

Définition 4.6. (Correspondance stricte, paresseuse)
Soient $(\mathcal{C}^D, \preccurlyeq_D, \mu_D)_{D\subset\mathcal{D}}$ et $(\mathcal{C}'^{D'}, \preccurlyeq'_{D'}, \mu'_{D'})_{D'\subset\mathcal{D}'}$ deux modèles de coût statique sur les langages respectifs $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$ et $L' : \mathcal{P}' \times \mathcal{D}' \to \mathcal{R}'$, et $T : \mathcal{P}' \to \mathcal{P}$, $T' : \mathcal{P} \to \mathcal{P}'$ des transformations de programmes.
La paire (T', T) forme une *correspondance stricte* entre $(L, \mathcal{C}^D, \preccurlyeq_D, \mu_D)_{D\subset\mathcal{D}}$ et $(L', \mathcal{C}'^{D'}, \preccurlyeq'_{D'}, \mu'_{D'})_{D'\subset\mathcal{D}'}$ ssi pour tout $p \in \mathcal{P}$, elle forme une connexion de Galois contractante

- entre $(\mathcal{P}, \equiv)$ et $(\mathcal{P}', \equiv)$
- entre $(Equiv_{\mathrm{str}}(p), \preccurlyeq_{Dom(p)})$ et $(Equiv_{\mathrm{str}}(T'(p)), \preccurlyeq'_{Dom(T'(p))})$

La paire (T', T) forme une *correspondance paresseuse* de $(L, \mathcal{C}^D, \preccurlyeq_D, \mu_D)_{D\subset\mathcal{D}}$ sur $(L', \mathcal{C}'^{D'}, \preccurlyeq'_{D'}, \mu'_{D'})_{D'\subset\mathcal{D}'}$ ssi pour tout $p \in \mathcal{P}$, elle forme une connexion de Galois contractante

- entre $(\mathcal{P}, \sqsupseteq)$ et $(\mathcal{P}', \sqsupseteq)$
- entre $(Equiv_{\mathrm{par}}(p), \preccurlyeq_{Dom(p)})$ et $(Equiv_{\mathrm{par}}(T'(p)), \preccurlyeq'_{Dom(T'(p))})$

Notons que si (T', T) est une correspondance entre L et L', alors (T, T') est une correspondance entre L' et L. Ce n'est pas le cas de la correspondance paresseuse.

La connexion de Galois contractante entre $(\mathcal{P}, \equiv)$ et $(\mathcal{P}', \equiv)$ signifie :

- $\forall p, q \in \mathcal{P} \quad p \equiv q \;\Rightarrow\; T'(p) \equiv T'(q)$ et $\forall p', q' \in \mathcal{P}' \quad p' \equiv q' \;\Rightarrow\; T(p') \equiv T(q')$
- $\forall p \in \mathcal{P} \quad T(T'(p)) \equiv p$ et $\forall p' \in \mathcal{P}' \quad T'(T(p')) \equiv' p'$

Dans le cas de la correspondance paresseuse, on a :

- $\forall p, q \in \mathcal{P} \quad p \sqsubseteq q \;\Rightarrow\; T'(p) \sqsubseteq T'(q)$ et $\forall p', q' \in \mathcal{P}' \quad p' \sqsubseteq q' \;\Rightarrow\; T(p') \sqsubseteq T(q')$
- $\forall p \in \mathcal{P} \quad T(T'(p)) \sqsupseteq p$ et $\forall p' \in \mathcal{P}' \quad T'(T(p')) \sqsupseteq p'$

Cette définition correspond au diagramme suivant.

$$\begin{tikzcd}
\mathcal{C} & (\mathcal{P} \times \mathcal{D}) \arrow[l, "\mu"'] \arrow[r, "L"] \arrow[d, shift right, "T'"'] & \mathcal{R} \\
\mathcal{C}' & (\mathcal{P}' \times \mathcal{D}') \arrow[l, "\mu'"'] \arrow[r, "L'"] \arrow[u, shift right, "T"'] & \mathcal{R}'
\end{tikzcd}$$

Comme le montre la proposition suivante, une correspondance permet de transporter des évaluations partielles d'un langage dans un autre.

Proposition 4.7. (Evaluation partielle et langages correspondants)
Soit (T', T) une correspondance stricte entre L et L' (resp. paresseuse de L sur L'). Alors pour tout programme $p \in \mathcal{P}$, et pour $P = Equiv_{\mathrm{str}}(p)$ et $P' = Equiv_{\mathrm{str}}(T'(p))$ (resp. $P = Equiv_{\mathrm{par}}(p)$ et $P' = Equiv_{\mathrm{par}}(T'(p))$),

- $T'(Evap_P(p)) \subset Evap_{P'}(T'(p))$
- $T'(Evap_P^{\mathrm{optF}}(p)) \subset Evap_{P'}^{\mathrm{optF}}(T'(p))$
- $T'(Evap_P^{\mathrm{optf}}(p)) \subset Evap_{P'}^{\mathrm{optf}}(T'(p))$
- $T'(Evap_P^{\min}(p)) \subset Evap_{P'}^{\min}(T'(p))$

On a des propositions symétriques pour T. En les combinants, on obtient :

- $T(T'(Evap_P(p))) \subset Evap_P(p)$
- $T(T'(Evap_P^{\mathrm{optF}}(p))) = Evap_P^{\mathrm{optF}}(p)$
- $T(T'(Evap_P^{\mathrm{optf}}(p))) \subset Evap_P^{\mathrm{optf}}(p)$
- $T(T'(Evap_P^{\min}(p))) \subset Evap_P^{\min}(p)$

Pour plus de lisibilité, nous n'avons pas noté les indices $\preccurlyeq$ et $\preccurlyeq'$, qui ne sont pas ambigus.

La démonstration de cette proposition, laissée en annexe (cf. §D.4), repose principalement sur les relations entre optimalité et connexion de Galois contractante (prop. 4.6), ainsi sur les lignes 5 et 7 du tableau 4.2. L'évaluation partielle optimale faible, qui fait défaut à ce tableau, nécessite un retour à la définition et un développement spécifique.

Cette proposition nous permet d'obtenir des évaluations partielles d'un programme p de L au moyen de résultats connus dans L'. En effet, si l'on peut trouver dans L' une évaluation partielle p' (resp. optimale forte, optimale faible, minimale) de $T(p)$, alors $T(p')$ est une évaluation partielle de p (resp. optimale forte, optimale faible, minimale).

Notons que l'égalité des évaluations partielles de $T(T'(p))$ et p n'est garantie que dans le cas de l'évaluation partielle optimale forte : $Evap^{\mathrm{optF}}(T(T'(p))) = Evap^{\mathrm{optF}}(p)$. On peut aussi remarquer que si $Evap^{\min}(p)$ est non-vide (résultat que l'on peut par exemple obtenir grâce à la proposition 4.5), alors $Evap^{\min}(T'(p))$ est également non-vide. Autrement dit, il y a des « chances » de trouver dans L' une évaluation partielle minimale p' de $T'(p)$, et l'on sait alors que $T(p') \in Evap^{\min}(p)$. Cette propriété est aussi vérifiée par les évaluations partielles optimale forte et faible.

Nous rencontrerons le cas particulier suivant, où les langages partagent les ensembles de valeurs sémantiques et de coûts.

$$\begin{tikzcd}
& \mathcal{P} \arrow[dl, "\mu"'] \arrow[dr, "L"] \arrow[dd, shift right, "T'"'] & \\
(\mathcal{C} \leftarrow \mathcal{D}) & & (\mathcal{D} \to \mathcal{R}) \\
& \mathcal{P}' \arrow[ul, "\mu'"] \arrow[ur, "L'"'] \arrow[uu, shift right, "T"'] &
\end{tikzcd}$$

Considérons le cas où la sémantique L' est injective, ce qui signifie qu'il n'y a (au plus) qu'une seule manière de coder dans $\mathcal{P}'$ une fonction de $\mathcal{D} \to \mathcal{R}$.

Proposition 4.8. (Modèle intermédiaire de la dénotation)
Soit $(\mathcal{C}^D, \preccurlyeq_D, \mu_D)_{D \subset \mathcal{D}}$ un modèle de coût réflexif de $L : \mathcal{P} \times \mathcal{D} \to \mathcal{R}$. Soient $L' : \mathcal{P}' \times \mathcal{D} \to \mathcal{R}$ un langage tel que $\hat{L}' : \mathcal{P}' \to (\mathcal{D} \to \mathcal{R})$, est injective, $\mu' : \mathcal{P}' \times \mathcal{D} \to \mathcal{C}$ une mesure de performance sur L', et $T : \mathcal{P}' \to \mathcal{P}$, $T' : \mathcal{P} \to \mathcal{P}'$ des transformations de programmes telles que :

- $L = L' \circ T'$ et $L' = L \circ T$
- $\mu \preccurlyeq \mu' \circ T'$ et $\mu' \preccurlyeq \mu \circ T$

Alors (T', T) est une correspondance stricte entre L et L'. De plus,

- $\forall p \in \mathcal{P} \quad T(T'(p)) \in Evap_{\mathrm{str}}^{\mathrm{optF}}(p)$

Pour tout programme $p \in \mathcal{P}$, le programme $T(T'(p))$ est une évaluation partielle stricte optimale forte de p.

L'idée de cette construction est d'envoyer les programmes de $p \in \mathcal{P}$ en quelque sorte sur leur *dénotation*, c'est-à-dire sur un objet qui représente $L(p)$: c'est ce qu'expriment l'équation $L = L' \circ T'$ et le fait que L' soit injective. La classe des programmes strictement équivalents à p est alors envoyée sur le singleton $\{T'(p)\}$. La *représentation* $T'(p) \in \mathcal{P}'$ de cette dénotation doit être choisie telle que l'on puisse trouver un μ' qui satisfasse $\mu \succcurlyeq \mu' \circ T'$ et $\mu' \succcurlyeq \mu \circ T$. En pratique, on cherche à ce que $\mu' \circ T'$ minimise directement le coût dans $\mathcal{P}$ et l'on a alors $\mu' \approx \mu \circ T$. Lorsque ces conditions sont remplies, la transformation $T \circ T'$ est un *évaluateur partiel optimal fort*.

4.6.3 Application.

Nous mettons en application ce résultat afin de construire un évaluateur partiel optimal de ZML. Pour cela, on établit un lien entre programmes, environnements et une représentation sur sous forme de graphes orientés sans cycle (DAG, voir par exemple [Sed88, Cou90b]), qui modélisent l'implémentation d'arbres avec partage.

Syntaxe. On considère $\mathcal{G}$ l'ensemble des graphes orientés sans cycle d'arêtes ordonnées, tels que les nœuds g (les sommets) de tout graphe $G \in \mathcal{G}$ sont étiquetés par :

- un constructeur $con \in \mathrm{Con}$
- un ensemble $V = \{var_1, \ldots, var_l\}$ de variables (éventuellement vide)

et sont tels que :

- les ensembles V qui annotent deux nœuds distincts sont disjoints deux à deux
- les racines du graphe portent une annotation $V \neq \varnothing$

Ces nœuds sont notés $g = con_V(g_1, \ldots, g_n)$ où $g_1, \ldots, g_n$ est la séquence des nœuds fils, ordonnée par les arêtes, et $\lceil g \rceil$ le nombre de nœuds pères de g.

Sémantique. Notons qu'à tout nœud $g \in G$ correspond une valeur $v = \mathrm{val}(g) \in \mathrm{Val}$ qui est l'arbre (fini car G n'a pas de cycle) issu de g. On définit la fonction $L' : \mathcal{G} \to \mathrm{Env}$ par (cela a un sens car les V sont disjoints) :

- $L'(G) = \{var \mapsto v \mid g = con_V(g_1, \ldots, g_n) \in G,\ var \in V,\ v = \mathrm{val}(g)\}$

L'équivalence opérationnelle sur $\mathcal{G}$ est : $G_1 \equiv G_2$ ssi $L'(G_1) = L'(G_2)$. En fait, il y a une correspondance biunivoque entre les graphes de $\mathcal{G}$ et les environnements de Env ; l'équivalence $\equiv$ sur $\mathcal{G}$ est donc l'égalité.

Modèle d'exécution. On munit ensuite $\mathcal{G}$ d'une mesure de coût $\mu' : \mathcal{G} \to \mathcal{C}$ de la manière suivante :

- $\forall G \in \mathcal{G} \quad \mu'(G) = \sum_{g \in G} \mu'(g)$ avec si $g = con_V(g_1, \ldots, g_n)$ et $m = \lceil g \rceil$ et $l = |V|$:
- $\mu'(g) = \begin{cases} \min(m\, c_{con_n}, c_{valbind} + c_{con_n} + m\, c_{var}) & \text{si } V = \varnothing \\ l\, c_{valbind} + c_{con_n} + (l - 1 + m)\, \min(c_{con_n}, c_{var}) & \text{si } V \neq \varnothing \end{cases}$

L'affectation de ces coûts prépare la traduction inverse, des graphes vers les programmes optimaux, qui est donnée plus bas. Le cas $V = \varnothing$ correspond au cas où aucune variable de $E = L'(G)$ ne lie $v = \mathrm{val}(g)$; on a alors le choix entre construire m nouvelles instances de v, ou bien n'en construire qu'une, que l'on lie dans un environnemment local, et que l'on réutilise ensuite en accédant m fois à une variable. Dans le cas $V \neq \varnothing$, la liaison dans l'environnement est obligatoire ; il faut $l + m$ productions de v, dont une au moins par construction explicite ; on choisit alors entre accéder à une variable et construire de nouveau v. On définit également la mesure de coût $\mu'' : \mathrm{Env} \to \mathcal{C}$ par $\mu''(E) = \mu'(L'^{-1}(E))$.

Traduction dans les graphes. Toute valeur $v \in \mathrm{Val}$ est représentée par un arbre étiqueté avec des constructeurs $con \in \mathrm{Con}$. À tout environnement $E = \{var_1 \mapsto v_1, \ldots, var_N \mapsto v_N\}$ correspond le graphe orienté sans cycle d'arêtes ordonnées qui représente la forêt des valeurs $v_1, \ldots, v_N$. On définit une traduction $T' : \mathcal{P} \to \mathcal{G}$ qui à tout programme $prog \in \mathcal{P}$, de valeur $E = Zml(prog)$, associe le graphe $G \in \mathcal{G}$ représenté par E, dont les nœuds $g = con_V(g_1, \ldots, g_n)$ sont annotés par :

- $V = \{var \in \mathrm{Var} \mid E(var) = \mathrm{val}(g)\}$

Notons que l'on a $E = Zml(prog) = L'(T'(prog))$, c'est-à-dire $Zml = L' \circ T'$.

Traduction dans les programmes. Réciproquement, à tout graphe $G \in \mathcal{G}$, on associe le programme $prog = T(G)$ déterminé par la transformation $T : \mathcal{G} \to \mathrm{Prog}$ suivante. On commence par faire un tri tolopologique (voir par exemple [Sed88]) des nœuds de G (c'est possible car G est un graphe sans cycle). On renverse ensuite la séquence de nœuds obtenus pour former une séquence s_G. Notons $()$ la séquence vide et $(g \mid s)$ la séquence constituée de g suivi des éléments de la séquence s. Nous nous donnons également $(var_v)_{v \in \mathrm{Val}}$ un ensemble des variables indicées par les valeurs, toutes différentes entre elles, et disjointes des ensembles de variables qui apparaissent dans G (en pratique un *nouveau* symbole de variable est généré à la demande). On définit ensuite les transformations $T_{exp} : \mathcal{G} \to \mathrm{Exp}$ et $T_{decs} : \mathcal{G}^* \to \mathrm{Decs}$ ainsi : pour tout $g = con_V(g_1, \ldots, g_n)$ et $m = \lceil g \rceil$,

- $T_{exp}(g) = \begin{cases} var_{\mathrm{val}(g)} & \text{si } V = \varnothing \text{ et } c_{valbind} + c_{con_n} + m\, c_{var} \preccurlyeq c_{con_n} \\ con(T_{exp}(g_1), \ldots, T_{exp}(g_n)) & \text{si } V = \varnothing \text{ et } c_{valbind} + c_{con_n} + m\, c_{var} \succ m\, c_{con_n} \\ var & \text{si } var \in V \neq \varnothing \end{cases}$
- $T_{decs}() =$ *déclaration vide*
- $T_{decs}(g|s) = \begin{cases} \texttt{local val } var_1 \texttt{ = } con(T_{exp}(g_1), \ldots, T_{exp}(g_m)) \texttt{ in } T_{decs}(s) \texttt{ end} \\ \qquad \text{si } V = \varnothing \text{ et } c_{valbind} + c_{con_n} + m\, c_{var} \preccurlyeq m\, c_{con_n} \\ T_{decs}(s) \\ \qquad \text{si } V = \varnothing \text{ et } c_{valbind} + c_{con_n} + m\, c_{var} \succ m\, c_{con_n} \\ \texttt{val } var_1 \texttt{=} con(T_{exp}(g_1), \ldots, T_{exp}(g_n)) \texttt{;val } var_2 \texttt{=} var_1 \texttt{;} \ldots \texttt{;val } var_l \texttt{=} var_1 \texttt{;} T_{decs}(s) \\ \qquad \text{si } V = \{var_1, \ldots, var_l\} \neq \varnothing \end{cases}$

L'ordre topologique inverse garantit que l'accès à une variable est toujours postérieur à sa liaison dans l'environnement courant. La transformation T est finalement définie par $T(G) = T_{decs}(s_G)$. On peut démontrer que pour tout graphe G, représentant un environnement $E = L'(G)$, le programme $T(G)$ est tel que $Zml(T(G)) = E$. Autrement dit, $L' = Zml \circ T$.

Transport de la mesure de performance. Soit $prog$ un programme et $G = T'(prog)$. Pour tout valeur v qui apparait dans la preuve π de l'éxécution $Zml(prog) = E$, on note $\mu(\pi, v)$ la participation de v dans le coût total $\mu(prog) = \sum_{v \in \mathrm{Val}} \mu(\pi, v)$:

- $\mu(\pi, v) = i\, c_{con_n} + j\, c_{var} + k\, c_{valbind}$

où i, j, k sont le nombre d'occurrences respectives dans π des règles (ρ_{con}), (ρ_{var}) et $(\rho_{valbind})$ où figure v. Soit $g = con_V(g_1, \ldots, g_n)$ le nœud qui représente v dans G, $m = \lceil g \rceil$, et $l = |V|$. Les contraintes suivantes sur $i, j, k, l, m \in \mathbb{N}$ sont toujours vérifiées :

- $m \leqslant i + j - k$ (l'environnement conserve moins d'occurrences d'une valeur que l'exécution)
- $1 \leqslant i$ (toute valeur qui apparaît dans l'environnement est au moins produite une fois)

- $1 \leqslant j \;\Rightarrow\; 1 \leqslant k$ (si une valeur est produite par l'accès à une variable, elle a nécessairement été liée)
- $l \leqslant k$ (l'environnement conserve moins de liaisons d'une valeur globale que l'exécution)

On étudie le signe de $\delta = \mu(\pi, v) - \mu'(g)$. Si $V = \varnothing$, c.-à-d. si $l = 0$,

- Si $m\, c_{con_n} \succcurlyeq c_{valbind} + c_{con_n} + m\, c_{var} = \mu'(g)$, alors $\delta = (i-1)\, c_{con_n} + (j-m)\, c_{var} + (k-1)\, c_{valbind}$. Notons alors que $c_{con_n} \succcurlyeq c_{var}$. En effet, dans le cas contraire, si $c_{con_n} \prec c_{var}$ alors $m\, c_{con_n} \prec m\, c_{var}$ car $m \geqslant 1$ puisque $V = \varnothing$, donc $c_{con_n} \prec m\, c_{var} + c_{valbind} + c_{con_n}$, ce qui contredit l'hypothèse.
  - Si $1 \leqslant j$ alors $0 \leqslant k-1$ donc $\delta \succcurlyeq (i-1)\, c_{con_n} + (j-m)\, c_{var}$, or $c_{var} \preccurlyeq c_{con_n}$ et $0 \leqslant i-1$ donc $\delta \succcurlyeq (i-1)\, c_{var} + (j-m)\, c_{var} = (i+j-m-1)\, c_{var}$, or $m \leqslant i+j-k$ donc $i+j-m-1 \geqslant k-1 \geqslant 0$, donc $\delta \succcurlyeq 0$.
  - Si $j = 0$ alors $\delta = (i-1)\, c_{con_n} - m\, c_{var} + (k-1)\, c_{valbind}$, or $-c_{valbind} + (m-1)\, c_{con_n} \succcurlyeq m\, c_{var}$, donc $\delta \succcurlyeq (i-m)\, c_{con_n} + k\, c_{valbind}$, or $m \leqslant i-k$, donc $0 \leqslant k \leqslant i-m$, et $\delta \succcurlyeq k\, c_{con_n} + k\, c_{valbind} \succcurlyeq 0$.
- Si $c_{valbind} + c_{con_n} + m\, c_{var} \succcurlyeq m\, c_{con_n} = \mu'(g)$, alors $\delta = (i-m)\, c_{con_n} + j\, c_{var} + k\, c_{valbind}$.
  - Si $c_{con_n} \preccurlyeq c_{var}$ alors $\delta \succcurlyeq (i-m)\, c_{con_n} + j\, c_{con_n} + k\, c_{valbind}$, or $m \leqslant i+j-k$, donc $0 \leqslant k \leqslant i+j-m$, donc $\delta \succcurlyeq k\, c_{con_n} + k\, c_{valbind} \succcurlyeq 0$.
  - Si $c_{var} \preccurlyeq c_{con_n}$:
    - Si $1 \leqslant j$ alors $1 \leqslant k$, donc $\delta = c_{valbind} + c_{con_n} + m\, c_{var} + (i-m-1)\, c_{con_n} + (j-m)\, c_{var} + (k-1)\, c_{valbind} \succcurlyeq (i-1)\, c_{con_n} + (j-m)\, c_{var} + (k-1)\, c_{valbind} \succcurlyeq (i-1+j-m)\, c_{var} + (k-1)\, c_{valbind}$, or $0 \leqslant k-1 \leqslant i+j-m-1$, donc $\delta \succcurlyeq (k-1)\, c_{var} + (k-1)\, c_{valbind} \succcurlyeq 0$.
    - Si $j = 0$ alors $m \leqslant i-k$, donc $0 \leqslant k \leqslant i-m$, donc $\delta \succcurlyeq 0$.

Si maintenant $V \neq \varnothing$, c.-à-d. $1 \leqslant l$,

- Si $c_{var} \preccurlyeq c_{con_n}$ alors $\mu'(g) = l\, c_{valbind} + c_{con_n} + (l-1+m)\, c_{var}$ et $\delta = (i-1)\, c_{con_n} + (j-l+1-m)\, c_{var} + (k-l)\, c_{valbind} \succcurlyeq (i-1+j-l+1-m)\, c_{var} + (k-l)\, c_{valbind}$ car $i \geqslant 1$, or $0 \leqslant k-l \leqslant i+j-m-l$, donc $\delta \succcurlyeq 0$.
- Si $c_{con_n} \preccurlyeq c_{var}$ alors $\mu'(g) = l\, c_{valbind} + (l+m)\, c_{con_n}$ et $\delta = (i-l-m)\, c_{con_n} + j\, c_{var} + (k-l)\, c_{valbind} \succcurlyeq (i+j-l-m)\, c_{var} + (k-l)\, c_{valbind}$, or $0 \leqslant k-l \leqslant i+j-m-l$, donc $\delta \succcurlyeq 0$.

Dans chaque cas, on a $\delta \succcurlyeq 0$. Par conséquent $\mu'(g) \preccurlyeq \mu(\pi, v)$ et $\mu'(T'(prog)) = \mu'(G) \preccurlyeq \mu(prog)$. On peut aussi montrer que pour tout graphe $G \in \mathcal{G}$, on a $\mu(T(G)) = \mu'(G)$.

Évaluateur partiel optimal. Nous sommes dans le cadre de la proposition 4.8 car :

- L' est une bijection
- $Zml = L' \circ T'$ et $L' = Zml \circ T$
- $\mu \succcurlyeq \mu' \circ T'$ et $\mu' = \mu \circ T$

Par conséquent $\boldsymbol{evap} = T \circ T'$ est un évaluateur partiel optimal. On peut noter que les constructions `let` et `and` sont superflues car elles n'interviennent pas dans les évaluations partielles optimales calculées par $\boldsymbol{evap}$.

Bien que cet exemple soit élémentaire (pour le langage, comme pour le modèle d'exécution), la construction d'un évaluateur partiel optimal n'est pas très simple[7]. Nous n'avons d'ailleurs pas donné toutes les preuves de

[7] Si l'on ne considère plus que E_0 est vide (cf. §4.6.1, sémantique) mais qu'il est un paramètre du problème, la situation devient plus compliquée. Par exemple, pour $E_0 = \{x \mapsto A\}$ et $E = \{x \mapsto B, y \mapsto F(A,B), z \mapsto G(A,B)\}$ on a, en cas de partage, des programmes optimaux de structure très différente :

```
local                     local
  val aux = x               val aux = B
in                        in
  val x = B                 val x = aux
  val y = F(aux,x)          and y = F(x,aux)
  val z = G(aux,x)          and z = G(x,aux)
end                       end
```

nos assertions. On mesure d'autant la difficulté de l'évaluation partielle optimale dans des cas plus complexes.

Il faut néanmoins noter que, dans l'esprit de la section §4.7, une implémentation optimale allouerait en fait *statiquement* les valeurs calculées par une expression constante[8].

4.7 Évaluation partielle et compilation optimale.

La définition 3.10 indique comment définir la performance d'un langage compilé : la performance d'un programme source p_s est celle de son code compilé $p_c = T(p_s)$. On peut ainsi effectuer l'évaluation partielle d'un langage source L_s selon les critères de performance d'un langage cible L_c.

Il existe alors deux évaluations partielles : l'une dans L_s, l'autre dans L_c. Cependant, ce n'est pas parce qu'une évaluation partielle q_s d'un programme p_s de L_s est optimale ou minimale que c'est également le cas de sa compilation $q_c = T(q_s)$ dans L_c.

Considérons par exemple le modèle de performance fourni par l'implémentation de la section §2.2. Le fait que les programmes `nand1` et `nand2` soient minimaux pour $\mu_{\mathrm{S.Nat}}$ avec le coût absolu et optimaux avec le coût moyen ne signifie pas que leur compilation l'est aussi pour μ_{Mach}. Le programme `nand1` a été compilé ainsi (cf. §2.2.2) :

```
nand1 : MOV BP,SP
        MOV AX,(BP+4);  CMP AX,0;  JZ lab3
        MOV AX,(BP+2);  CMP AX,0;  JZ lab1
        MOV AX,0;  JMP lab2
lab1 :  MOV AX,1
lab2 :  JMP lab4
lab3 :  MOV AX,1
lab4 :  RET 4
```

Son coût d'exécution par μ_{Mach} est :

$$c_{\texttt{MOV}\,reg,reg} + c_{\texttt{MOV}\,reg,(reg+depl)} + c_{\texttt{CMP}\,reg,imm} + c_{\texttt{MOV}\,reg,imm} + c_{\texttt{RET}\,depl} +$$
$$\begin{cases} \text{si } (\texttt{SP+4}) = 0 & \text{alors } c_{\texttt{JZ}_{\texttt{ZF=1}}\,addr} \\ \text{si } (\texttt{SP+4}) = 1 \text{ et } (\texttt{SP+2}) = 1 & \text{alors } c_{\texttt{MOV}\,reg,(reg+depl)} + c_{\texttt{CMP}\,reg,imm} + 2\,(c_{\texttt{JZ}_{\texttt{ZF=0}}\,addr} + c_{\texttt{JMP}\,addr}) \\ \text{si } (\texttt{SP+4}) = 1 \text{ et } (\texttt{SP+2}) = 0 & \text{alors } c_{\texttt{MOV}\,reg,(reg+depl)} + c_{\texttt{CMP}\,reg,imm} + c_{\texttt{JZ}_{\texttt{ZF=0}}\,addr} + c_{\texttt{JZ}_{\texttt{ZF=1}}\,addr} \\ & \quad + c_{\texttt{JMP}\,addr} \end{cases}$$

En revanche, le programme Mach suivant économise des branchements inutiles.

[8] C'est le cas par exemple du compilateur POLYML version 2.05 qui construit une fois pour toutes les structures de données correspondantes au moment de la compilation ; leur « évaluation » est alors immédiate, de temps constant, infime, et de coût mémoire dynamique nul. En revanche, nous ne connaissons pas d'implémentation qui réalise le partage des expressions.

```
nandopt : MOV BP,SP
          MOV AX,(BP + 4);  CMP AX,0;  JZ lab
          MOV AX,(BP + 2);  CMP AX,0;  JZ lab
          MOV AX,0;  RET 4
lab :     MOV AX,1;  RET 4
```

Il a pour coût d'exécution :

$$c_{\texttt{MOV}\,reg,reg} + c_{\texttt{MOV}\,reg,(reg+depl)} + c_{\texttt{CMP}\,reg,imm} + c_{\texttt{MOV}\,reg,imm} + c_{\texttt{RET}\,depl} +$$
$$\begin{cases} \text{si } (\texttt{SP+4}) = 0 & \text{alors } c_{\texttt{JZ}_{\texttt{ZF}=1}\,addr} \\ \text{si } (\texttt{SP+4}) = 1 \text{ et } (\texttt{SP+2}) = 1 & \text{alors } c_{\texttt{MOV}\,reg,(reg+depl)} + c_{\texttt{CMP}\,reg,imm} + 2\,c_{\texttt{JZ}_{\texttt{ZF}=0}\,addr} \\ \text{si } (\texttt{SP+4}) = 1 \text{ et } (\texttt{SP+2}) = 0 & \text{alors } c_{\texttt{MOV}\,reg,(reg+depl)} + c_{\texttt{CMP}\,reg,imm} + c_{\texttt{JZ}_{\texttt{ZF}=0}\,addr} + c_{\texttt{JZ}_{\texttt{ZF}=1}\,addr} \end{cases}$$

Comparé à la compilation de `nand1`, ce programme est strictement meilleur pour le coût absolu, et par conséquent, grâce à la proposition 3.3, aussi strictement meilleur pour le coût moyen, qui est stable et discriminant (tabl. 3.3). La compilation de `nand1` n'est donc ni optimale, ni minimale, et c'est en fait le programme `nandopt` qui est minimal dans Mach pour le coût absolu, et optimal pour le coût moyen (si l'on se limite aux seules instructions définies pour Mach à la section §2.2.2).

Autrement dit, on perd en performance à vouloir optimiser dans L_s plutôt que dans L_c. La raison est que dans le premier cas on optimise sur $T(\mathcal{P}_s)$, et dans le second sur $\mathcal{P}_c \supset T(\mathcal{P}_s)$. Cet argument, qui remet en cause le bien-fondé du problème de l'évaluation partielle, sera repris dans la conclusion.

4.8 Essence des optimisations.

Pour clore ce chapitre, nous examinons de manière informelle les « grands » principes qui sous-tendent la majorité des optimisations. Nous examinons également quel sens donner à l'optimisation d'un ensemble d'exécutions.

4.8.1 Les principes.

Un langage de programmation associe deux concepts : *donnée* (les valeurs et les opérations sur les valeurs) et structure de *contrôle* (branchements, appels de routines). En pratique, une machine d'exécution (séquentielle) effectue les tâches suivantes.

- construction de données,
- déplacement de données,
- destruction de données,
- déplacement du point de contrôle.

Une optimisation réduit l'importance ou le nombre de tâches à effectuer au moment de l'exécution, tout en préservant le comportement opérationnel des programmes. Elle obéit aux principes suivants.

Ce qui doit (ou peut) être calculé, le faire avant l'exécution. Ce principe, qui déplace des calculs dans le temps (cf. §4.8.3), est à la base de la spécialisation (cf. §1.3) et de l'évaluation partielle généralisée (cf. §1.5) : une expression statique (indépendante des paramètres du programme) est remplacée par sa valeur[9] ; de même, une suite de branchements est réduite. Il y a généralement gain en termes de temps et d'espace mémoire dynamiques, quels que soient les paramètres du programme : le programme optimisé est meilleur que le programme original au sens du coût absolu (et donc du coût moyen). Cependant, ce n'est pas toujours vrai pour les données qui ne sont pas atomiques (cf. §4.6.1, optimisation).

Plus généralement, si l'on imagine l'exécution d'un programme comme celle d'un système de transition, ce principe consiste à recenser statiquement les futurs états dynamiques qui comporteront des composantes invariables et sur lesquelles il est possible de faire dès à présent des calculs. C'est ainsi par exemple qu'est éliminé le niveau d'interprétation d'un méta-interpréte (cf. §1.3.3).

Lors de l'exécution, calculer un minimum de fois. Ce principe contient en particulier l'élimination des calculs inutiles (les valeurs sont calculées zéro fois). Cela comprend notamment la réduction des mouvements de données (propagation des constantes, allocation des registres, de la pile, etc.) et l'élimination des contructions temporaires (supercompilation, déforestation, cf. §1.4).

Ce principe exprime aussi que chaque valeur calculée à l'exécution ne doit pas être recalculée inutilement ; c'est une abstraction de l'élimination des redondances. Il ne s'agit pas simplement de partager des sous-termes (cf. §4.6.1, optimisation), mais plus généralement de faire en sorte que toute valeur soit calculée un minimum de fois au cours d'une exécution. C'est l'objet d'un compromis que rencontrent les programmes incrémentaux : il vaut mieux recalculer certaines valeurs, et d'autres, les préserver afin de les réutiliser (cf. §4.8.3).

Données multiformes. Une autre source d'optimisation consiste à avoir plusieurs représentations concrètes d'une même donnée abstraite. Alternativement, on peut voir cela comme un quotient des résultats d'exécution par une relation d'équivalence. L'état courant d'une classe est un représentant déterminé par l'histoire de son calcul ou son usage futur (arbres non-équilibrés, matrices creuses, etc.). Dans le cas de l'évaluation partielle, cette notion peut être matérialisée dans un langage de haut niveau par l'emploi d'un type abstrait.

Dans le même ordre d'idées, on peut aussi penser aux modèles de compilation dans lesquels un même programme peut choisir à l'exécution parmi plusieurs implémentations la plus adpatée. Dans notre formalisation, la donnée d'une famille de pondération permet précisément d'obtenir ce type de comportement. À différents modes d'utilisation d'un programme correspondent différentes distributions. Il suffit de calculer, statiquement, des évaluations partielles optimales qui correspondent à chacune de ces distributions. Puis, dynamiquement, le programme doit déterminer de quelle distribution ses données sont le plus proches, et exécuter l'évaluation partielle correspondante.

4.8.2 Le compromis espace/temps.

Pour optimiser un programme, on peut être amener à faire croître sa taille arbitrairement, et donc immodérément. L'espace mémoire occupé peut aussi être consommé durant l'exécution dans le cas d'une compilation dynamique (automate, byte-code. . .). Un compromis entre l'espace et le temps est donc inévitable et indispensable.

[9] Tous les langages ne permettent pas la coercition directe d'une valeur en un élément de programme. Un programme PROLOG par exemple ne fait pas apparaître explicitement de valeurs, qui sont les séquences (éventuellement infinies) de substitutions ; il faut nécessairement les représenter par programme.

Par exemple, la transformation x`::(if` c `then` y `else` z`)` $\rightarrow$ `if` c `then` x`::`y `else` x`::`z lorsque x, y et z sont statiques (cf. §1.3.5, valeurs conditionellement statiques) permet de gagner du temps sur la construction des listes, qui peut être réalisée statiquement, mais génère un programme plus coûteux en place mémoire.

Autre exemple, le langage rationnel dénoté par l'expression régulière $(\mathsf{a}+\mathsf{b})^*\mathsf{a}(\mathsf{a}+\mathsf{b})^{n-1}$ est reconnu par un automate fini déterministe minimal de taille $\mathcal{O}(2^n)$, qui minimise également le temps $\mathcal{O}(l)$ nécéssaire pour accepter une chaîne de longeur l. Cependant, une version non-derterministe de cet automate ne nécessite qu'un espace $\mathcal{O}(n)$, pour un temps d'exécution simplement $\mathcal{O}(nl)$.

Cet exemple illustre accessoirement l'influence de résultats issus de la théorie des automates sur l'optimisation [Var94]. En effet, à divers types d'automates (fini, à pile, etc.) correspondend des classes de fonctions. L'existence d'automates minimaux et certaines propriétés de stabilité (par intersection, complémentaire, etc.) ont des retombées en terme d'optimisations. En particulier, des procédures de décision sur ces automates permettent la résolution statique de certaines expressions symboliques, dans l'esprit de l'évaluation partielle généralisée (cf. §1.5).

Le compromis espace/temps est au cœur des optimisations par évaluation partielle, qui bien souvent ne font qu'éliminer des niveaux d'interprétation : on éclate les fonctionnalités en les spécialisant, afin de gagner du temps à l'exécution. En pratique, il y a peu ou pas de contrôle. Si l'expansion des programmes est parfois bornée [Sah91a], on ne sait pas faire grand chose d'autre en cas de débordement que de s'arrêter au point où l'on en est. Par ailleurs, l'exemple phare de la compilation d'expressions régulières par évaluation partielle génère une fonction par état ; par rapport à un automate, le code est plus rapide d'un facteur linéaire, mais aussi plus gros d'un autre facteur linéaire.

4.8.3 Optimiser un ensemble d'exécutions.

La vie d'un programme se déroule sur plusieurs périodes : une période d'optimisation par évaluation partielle (partial evaluation time), une période de compilation (compilation time), et enfin la période d'exécution proprement dite (run time). Le coût de chacune de ces phases peut être *amorti* en cas d'utilisation répétée des objets qu'elles produisent (cf. §1.3.3).

C'est un point important en évaluation partielle : alors qu'il est généralement obligatoire de compiler un programme, l'optimisation par évaluation partielle reste un choix. Il faut savoir dans quelles circonstances cette opération est rentable.

En particulier, il est toujours possible d'améliorer des parties du programme original p qui ne seront en fait jamais atteintes par l'exécution sur certaines données ou même sur tout $Dom(p)$. Il y a alors perte systématique, indépendamment du modèle de performance : des ressources ont été consommées inutilement.

Optimisation statique.

Considérons un programme p devant être exécuté sur plusieurs données $d_1, \ldots, d_n$. Sans traitement préalable, la mesure de performance de l'exécution globale est $c_1 + \cdots + c_n$, où c_i représente le coût d'exécution de $p(d_i)$.

Si l'on optimise le programme p en un programme p', opération qui elle-même a un coût κ (que nous supposons homogène aux c_i), la même mesure donne pour p' un coût $c'_1 + \cdots + c'_n$, où c'_i représente le coût d'exécution de $p'(d_i)$. Il y a un gain global ssi $\kappa + c'_1 + \cdots + c'_n \preccurlyeq c_1 + \cdots + c_n$. Le coût de l'optimisation est en quelque sorte partagé par chacune des données, qui y trouve finalement son avantage.

Pour donner un ordre d'idée, supposons les coûts c_i et c'_i scalaires et constants, égaux respectivement à c et c', et tels que $c' \prec c$. L'optimisation de p en p' est avantageuse ssi $\kappa + nc' \preccurlyeq nc$, c'est-à-dire ssi $n \geqslant \kappa/(c - c')$. Dans le cas où $\kappa/(c - c') \leqslant 1$, il est avantageux d'optimiser le programme, même s'il ne sert qu'une fois ($n = 1$). Dans le cas contraire, l'opération n'est rentable qu'à partir de plusieurs utilisations.

Cette inégalité a aussi une importance pratique pour justifier l'intérêt de la recherche de meilleures évaluations partielles. En effet, comme nous l'avons vu à la section §4.5, le calcul d'une évaluation partielle optimale ou minimale — quand il est possible — est souvent extrêmement couteux. Toutefois, ce calcul n'est à faire qu'une seule fois, lorsque le programme est terminé et mis au point [10].

Optimisation dynamique.

La *mémoïsation* est une méthode dynamique pour simuler le gain procuré par ce partage au cours d'une seule exécution. Elle troque de l'espace dynamique contre du temps.

Les résultats de certains appels fonctionnels (il ne faut pas qu'interviennent d'effets de bord car sans cela on ne peut garantir que le même appel renverra le même résultat) au cours d'une même exécution sont dynamiquement stockés dans une table avec leurs paramètres d'appel. Lorsque la *mémo-fonction* est appelée de nouveau, on regarde dans cette table si les paramètres sont identiques à un appel déjà effectué auparavant ; si c'est le cas, la table contient également la valeur de retour et il n'est pas nécessaire d'exécuter une nouvelle fois la fonction. Cette méthode conduit parfois à de l'*évaluation partielle dynamique* [HG91].

La mémoïsation est en fait une des facettes de la *programmation dynamique* (voir par exemple [Sed88]). Recalculer ou stocker pour réutiliser est une des sources de compromis entre temps et espace dynamique.

Conclusion du chapitre.

Grâce aux notions définies dans les chapitres précédents, nous avons pu donner une définition formelle de l'évaluation partielle (déf. 4.3). Parce qu'il existe plusieurs notions d'optimalité (§4.2), il existe également plusieurs types d'évaluation partielle optimale (déf. 4.4) ; nous avons indiqué comment elles étaient reliées entre elles (prop. 4.4) et quels étaient leurs liens avec les constructions de coût (tabl. 4.2).

L'illustration de ces propositions sur les exemples de coût statique définis au chapitre précédent montre que l'ensemble de ces définitions forme un tout cohérent, à la fois formel et concret (§4.4). De plus, les relations entre propriétés permettent d'anticiper ou de mieux comprendre certains résultats.

Nous avons ensuite étudié les difficultés intrinsèques des évaluations partielles optimales et minimale, recensant des conditions d'existence (§4.5.2) et d'inexistence (§4.5.1). Beaucoup se présentent sous forme de méthodologies à appliquer à un problème spécifique. Nous avons toutefois pu obtenir un résultat formel ;

[10] C'est une tâche non-interactive qu'en pratique on peut lancer la nuit, sur les machines souvent moins chargées. C'est grâce à cela que les chiffres donnés dans la note de bas de page numéro 3 trouvent leur justification.

avec quelques hypothèses sur le domaine de coût, nous avons montré qu'il existait une évaluation partielle minimale pour tout programme de domaine fini (prop. 4.5). Nous avons également constaté l'échec ou les limites de plusieurs méthodes de construction explicite (§4.5.3) : procéder par composition est impossible, par transformations monotones, hasardeux, et par énumération, excessivement couteux.

Nous avons montré comment il était possible de résoudre des problèmes d'évaluation partielle, notamment minimale et optimale, en passant par l'intermédiaire d'un autre langage (prop. 4.7). Nous avons donné comme exemple un évaluateur partiel optimal d'expressions constantes pour SML (§4.6.3).

Par ailleurs, nous avons noté qu'une évaluation partielle optimale ne signifie pas que sa compilation l'est aussi. Nous reviendrons dans la conclusion de cette thèse sur cette pierre d'achoppement de l'évaluation partielle.

Enfin, nous avons recensé quelques principes de bases qui régissent les méthodes d'optimisation (§4.8). Nous avons aussi indiqué le sens et l'importance de l'optimisation sur un ensemble d'exécution (§4.8.3).

Le chapitre suivant étudie la correction de transformations qui permettent de construire des évaluations partielles.

Chapitre 5

Transformations et Correction

Les principales transformations de programmes employées en évaluation partielle sont le *pliage* et le *dépliage* (cf. §1.3.4, 1.4.3, 1.5.3, transformations). Cependant, elles ne sont pas toujours *correctes* (cf. §1.3.4, correction). Pour étudier leur correction, nous avons recours aux schémas de programmes récursifs (cf. §2.5).

Il faut toutefois noter que les schémas de programmes ne sont pas bien adaptés pour étudier la correction dans un langage de programmation *strict*. En effet, alors que dans un tel langage le dépliage est complet mais pas nécessairement valide (cf. §1.3.4, correction), le dépliage est par contre toujours correct pour des schémas de programmes (cf. §5.2.1). En fait, leur sémantique correspond (entre autres) à la stratégie de réduction de l'appel par nom (cf. §2.5.4) et ne peut donc être employée pour modéliser un langage de programmation strict.

C'est une des raisons pour lesquelles nous avons préconisé l'emploi de schémas *réguliers* qui, eux, ne posent pas ce type de problème (cf. §2.5.7). Néanmoins, dans un soucis de généralité, et aussi parce que cela peut être utile dans le cas d'un langage paresseux, nous étudions ici des conditions de correction pour des schémas *algébriques*, et non simplement réguliers. Par ailleurs, dans les manipulations de systèmes de réécriture qui suivent, les degrés de difficulté restent de toute façon voisins.

Organisation du chapitre.

§5.1 Nous donnons des définitions formelles qui concernent la *correction* des transformations de programme.

§5.2 Nous puisons ensuite dans la littérature plusieurs conditions pour qu'une transformation soit correcte, dont celle de *dépliage/pliage faible*, qui impose de faire d'abord des dépliages, puis des réécritures sémantiques, et enfin des pliages. Toute transformation se ramène ensuite à ce cas particulier pourvu que les règles sémantiques des lois algébriques soient linéaires à gauche *et* à droite.

§5.3 Ce dernier point met en jeu des propriétés de *commutativité* des systèmes de réécriture. Nous démontrons quelques lemmes et propositions nouveaux qui permettent de faire commuter des dérivations.

§5.4 Ces propositions montrent qu'une linéarité à gauche *ou* à droite suffit à ramener une transformation de pliage/dépliage quelconque à une transformation de dépliage/pliage faible.

§5.5 Enfin, une dernière section définit une *séquence de transformations* comme une suite de transformations élémentaires qui mettent en jeu pliages, dépliages, réécritures selon les lois algébriques, et définitions de fonctions auxiliaires. Nous montrons qu'elle peut toujours se ramener à une forme normale, sur laquelle nous disposons des résultats de correction de la section §5.4.

Le chapitre suivant s'intéresse aux algorithmes de transformation, et notamment à leur terminaison.

5.1 Correction d'une transformation.

La présentation des schémas de programme de la section §2.5 se cantonnait à la sémantique. Nous donnons ici les notions qui concernent la *correction d'une transformation*.

5.1.1 Aperçu.

Classe d'interprétations et équivalence. Un ensemble d'équations sur les symboles de base d'un schéma, dites *lois algébriques*, détermine une *classe équationnelle* d'interprétations, constituée précisément de l'ensemble des interprétations qui respectent ces lois. Par exemple, les interprétations usuelles sur les listes « à la LISP » appartiennent à la classe définie par les lois suivantes.

$$\begin{array}{lcl} \mathsf{car}(\mathsf{cons}(x,y)) & = & x \\ \mathsf{cdr}(\mathsf{cons}(x,y)) & = & y \\ \mathsf{if}(\mathsf{nil},x,y) & = & y \\ \mathsf{if}(\mathsf{cons}(x',y'),x,y) & = & x \end{array}$$

Ces équations décrivent en fait des opérations *paresseuses*. Une classe équationnelle stricte est par exemple :

$$\begin{array}{lcl} \mathsf{car}(\mathsf{cons}(x,d)) & = & x \quad \text{si} \quad d \neq \bot \\ \mathsf{car}(\mathsf{cons}(x,\bot)) & = & \bot \\ \mathsf{cdr}(\mathsf{cons}(d,y)) & = & y \quad \text{si} \quad d \neq \bot \\ \mathsf{cdr}(\mathsf{cons}(\bot,y)) & = & \bot \end{array}$$

Deux schémas de programme sont *équivalents* selon une classe d'interprétations s'ils ont la même sémantique pour tous les éléments de la classe.

Correction d'une transformation. Une transformation est *correcte* selon une classe d'interprétations si elle transforme tout schéma en un schéma équivalent. Reprenons un des exemples de spécialisation (cf. §1.3.2) en terme de schémas.

$$\begin{array}{lcl} \mathbf{append}(x,y) & = & \mathsf{if}(x,\mathsf{cons}(\mathsf{car}(x),\mathbf{append}(\mathsf{cdr}(x),y)),y) \\ \mathbf{append12}(y) & = & \mathbf{append}(\mathsf{cons}(1,\mathsf{cons}(2,\mathsf{nil})),y) \end{array}$$

Pour toute interprétation appartenant à la classe équationnelle ci-dessus, les transformations suivantes sont correctes.

$$\begin{array}{lcl} \mathbf{append12}(y) & = & \underline{\mathbf{append}}(\mathsf{cons}(1,\mathsf{cons}(2,\mathsf{nil})),y) \\ & = & \underline{\mathsf{if}(\mathsf{cons}}(\ldots),\mathsf{cons}(\underline{\mathsf{car}(\mathsf{cons}}(1,\mathsf{cons}(2,\mathsf{nil}))),\mathbf{append}(\underline{\mathsf{cdr}(\mathsf{cons}}(1,\mathsf{cons}(2,\mathsf{nil}))),y)),y) \\ & = & \mathsf{cons}(1,\underline{\mathbf{append}}(\mathsf{cons}(2,\mathsf{nil}),y)) \\ & = & \mathsf{cons}(1,\underline{\mathsf{if}(\mathsf{cons}}(2,\mathsf{nil}),\mathsf{cons}(\underline{\mathsf{car}(\mathsf{cons}}(2,\mathsf{nil})),\mathbf{append}(\underline{\mathsf{cdr}(\mathsf{cons}}(2,\mathsf{nil})),y)),y)) \\ & = & \mathsf{cons}(1,\mathsf{cons}(2,\underline{\mathbf{append}}(\mathsf{nil},y))) \\ & = & \mathsf{cons}(1,\mathsf{cons}(2,\underline{\mathsf{if}(\mathsf{nil}},\mathsf{cons}(\mathsf{car}(\mathsf{nil}),\mathbf{append}(\mathsf{cdr}(\mathsf{nil}),y)),y))) \\ & = & \mathsf{cons}(1,\mathsf{cons}(2,y)) \end{array}$$

L'équation de **append12** a subi ici des transformations de *réécriture suivant les lois algébriques* et de *dépliage*.

5.1.2 Définitions.

Les définitions suivantes font suite à celles données à la section §2.5.2. Elles sont en majeure partie tirées de [Cou90a].

Classe d'interprétation. Soit $\boldsymbol{C}$ une *classe* (un ensemble) de F-interprétation.

- $\boldsymbol{M}$ est $\boldsymbol{C}$*-initiale* ssi pour tout $\boldsymbol{D} \in \boldsymbol{C}$ il existe un homomorphisme de $\boldsymbol{M}$ dans $\boldsymbol{D}$.
- L'interprétation $\boldsymbol{M}_\Omega^\infty(F)$ est initiale vis à vis de l'ensemble de toutes les F-interprétations.

Comparaison de schémas. Soient Σ et Σ' deux systèmes d'équations sur F avec l'ensemble d'inconnues Φ, et $\boldsymbol{C}$ une classe de F-interprétations.

- $\Sigma \leqslant_{\boldsymbol{C}} \Sigma'$ ssi $\forall \boldsymbol{D} \in \boldsymbol{C} \quad \Sigma_{\boldsymbol{D}} \leqslant \Sigma'_{\boldsymbol{D}}$
- $\Sigma \equiv_{\boldsymbol{C}} \Sigma'$ ssi $\Sigma \leqslant_{\boldsymbol{C}} \Sigma'$ et $\Sigma \geqslant_{\boldsymbol{C}} \Sigma'$

On omet l'indice $\boldsymbol{C}$ lorsque c'est l'ensemble de toutes les F-interprétations, ou bien lorsque $\boldsymbol{C} = \boldsymbol{M}_\Omega^\infty(F)$.

Classe équationnelle. Soit R un sous-ensemble de $M(F,X) \times M(F,X)$. L'ensemble R est un ensemble d'équations qui expriment les lois algébriques des fonctions représentées par les opérateurs de F.

- La *classe équationnelle* $\boldsymbol{C}(R)$ définie par R est la classe de toutes les F-interprétations $\boldsymbol{D}$ tels que $t_{\boldsymbol{D}} = t'_{\boldsymbol{D}}$ pour tout $(t,t') \in R$ (voir aussi [Cou86, Cou90a, Gue81]).
- $\boldsymbol{C}(R)$ est l'ensemble de toutes les interprétations ssi R est un sous-ensemble, éventuellement vide, de la relation égalité sur $M(F,X)$.
- On note $\Sigma \leqslant_R \Sigma'$ pour $\Sigma \leqslant_{\boldsymbol{C}(R)} \Sigma'$, et $\Sigma \equiv_R \Sigma'$ pour $\Sigma \equiv_{\boldsymbol{C}(R)} \Sigma'$. On omet l'indice R lorsque $\boldsymbol{C}(R)$ est l'ensemble de toutes les F-interprétations.

Transformation et correction. Soit $\boldsymbol{C}$ une classe de F-interprétations.

- Une *transformation* $\boldsymbol{trans}$ est une relation entre systèmes d'équations : $\Sigma\ \boldsymbol{trans}\ \Sigma'$. Le schéma Σ' est aussi appelé une *transformation de* Σ *par* $\boldsymbol{trans}$.
- Une transformation de Σ par $\boldsymbol{trans}$ est $\boldsymbol{C}$*-correcte* ssi pour tout Σ' tel que $\Sigma\ \boldsymbol{trans}\ \Sigma'$, alors $\Sigma \equiv_{\boldsymbol{C}} \Sigma'$.
- Une transformation $\boldsymbol{trans}$ est $\boldsymbol{C}$*-correcte* ssi pour tous systèmes d'équations Σ et Σ', si $\Sigma\ \boldsymbol{trans}\ \Sigma'$, alors $\Sigma \equiv_{\boldsymbol{C}} \Sigma'$.

Une transformation *correcte* est une transformation $\boldsymbol{M}_\Omega^\infty(F)$*-correcte*.

5.2 Principales transformations.

Soit $\Sigma = \{\varphi_i(x_{i,1},\ldots,x_{i,n_i}) = t_i \mid i \in [N]\}$ un système d'équations sur F avec les inconnues Φ. Nous notons également Σ le système de réécriture $\{\langle\varphi_i(x_{i,1},\ldots,x_{i,n_i}) \to t_i\rangle \mid i \in [N]\}$. Soit $\mathcal{E}$ un ensemble d'équations sur $M(F,X)$ exprimant les lois algébriques des fonctions représentées par les opérateurs de F. On oriente les équations $\alpha = \beta$ de $\mathcal{E}$ pour former un système de réécriture $R \subset M(F,X) \times M(F,X)$ constitué des *régles sémantiques (suivant les lois algébriques)* $\langle \alpha \to \beta \rangle$. Il faut noter que nous n'imposons pas la contrainte $Var(\alpha) \supset Var(\beta)$ (cf. §N.17). On définit :

$$\begin{aligned} \underset{\Sigma,R}{\longrightarrow} &= \underset{\Sigma}{\longrightarrow} \cup \underset{R}{\longrightarrow} \cup \underset{R}{\longleftarrow} \\ \underset{\Sigma,R}{\longleftrightarrow} &= \underset{\Sigma}{\longrightarrow} \cup \underset{\Sigma}{\longleftarrow} \cup \underset{R}{\longrightarrow} \cup \underset{R}{\longleftarrow} \end{aligned}$$

5.2.1 Transformations fondamentales.

Le système d'équations $\Sigma' = \{\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) = t'_i \mid i \in [N]\}$ est obtenu respectivement par *réécriture selon les lois algébriques* (rewriting according to the algebraic laws), *dépliage* (unfold), *pliage* (fold), *pliage/dépliage* (fold/unfold) du système d'équations $\Sigma = \{\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) = t_i \mid i \in [N]\}$ ssi :

$$\begin{array}{lll} \Sigma\ \mathbf{rewr}_R\ \Sigma' & \text{ssi} & \forall i \in [N] \quad t_i \overset{*}{\underset{R}{\longleftrightarrow}} t'_i \\ \Sigma\ \mathbf{unf}_R\ \Sigma' & \text{ssi} & \forall i \in [N] \quad t_i \overset{*}{\underset{\Sigma,R}{\longrightarrow}} t'_i \\ \Sigma\ \mathbf{fld}_R\ \Sigma' & \text{ssi} & \forall i \in [N] \quad t_i \overset{*}{\underset{\Sigma,R}{\longleftarrow}} t'_i \\ \Sigma\ \mathbf{ufld}_R\ \Sigma' & \text{ssi} & \forall i \in [N] \quad t_i \overset{*}{\underset{\Sigma,R}{\longleftrightarrow}} t'_i \end{array}$$

Nous avons les propositions suivantes [Cou90a, prop. 7.1] :

$$\begin{array}{llll} \text{si} & \Sigma\ \mathbf{rewr}_R\ \Sigma' & \text{alors} & \Sigma \equiv_R \Sigma' \\ \text{si} & \Sigma\ \mathbf{unf}_R\ \Sigma' & \text{alors} & \Sigma \equiv_R \Sigma' \\ \text{si} & \Sigma\ \mathbf{fld}_R\ \Sigma' & \text{alors} & \Sigma \geqslant_R \Sigma' \\ \text{si} & \Sigma\ \mathbf{ufld}_R\ \Sigma' & \text{alors} & \Sigma \geqslant_R \Sigma' \end{array}$$

L'inégalité $\Sigma \geqslant_R \Sigma'$ signifie intuitivement que la transformation préserve la correction partielle, mais pas la terminaison, c'est-à-dire que les fonctions $\varphi \in \Phi$ vérifient $\varphi_{\Sigma,\boldsymbol{D}} \geqslant \varphi_{\Sigma',\boldsymbol{D}}$ pour tout $\boldsymbol{D} \in \boldsymbol{C}(R)$: $\varphi_{\Sigma',\boldsymbol{D}}$ est moins définie que $\varphi_{\Sigma,\boldsymbol{D}}$. On dit également *dépliage/pliage* (unfold/fold) pour pliage/dépliage.

5.2.2 Pliage/dépliage restreint.

Comme le montrent les exemples des sections §1.3 et §1.4, la transformation de pliage est indispensable en évaluation partielle. Cependant, à la différence du dépliage, elle n'est pas toujours correcte. Courcelle impose des contraintes sur $\mathbf{ufld}_R$ afin d'assurer la correction de certains pliages [Cou86, Cou90a].

Pour tout $I \subset [N]$, on note $\Sigma{\uparrow}I$ l'ensembles des équations de $\Sigma = \{\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) = t_i \mid i \in [N]\}$ portant sur les inconnues $(\varphi_i)_{i \in I}$. Le système d'équations $\Sigma' = \{\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) = t'_i \mid i \in [N]\}$ est obtenu par *pliage/dépliage restreint* (restricted fold/unfold) de Σ, noté $\Sigma\ \mathbf{rufld}_R\ \Sigma'$ ssi il existe un sous-ensemble d'indices $I \subset [N]$ tel que :

- $\forall i \in I \quad t_i = t'_i$
- $\forall i \in [N] \setminus I \quad t_i \overset{*}{\underset{\Sigma\uparrow I,R}{\longleftrightarrow}} t'_i$

Cette définition stipule que les équation de Σ utilisées pour la transformation (c'est-à-dire $\Sigma{\uparrow}I$) ne sont pas elles-mêmes transformées. Cette transformation est correcte :

- Proposition [Cou90a, prop. 7.2] : si Σ **rufld**$_R$ Σ' alors $\Sigma \equiv_R \Sigma'$

Cependant, cette contrainte sur les pliages est trop forte pour un usage en évaluation partielle : les exemples des sections §1.3 et §1.4 nécessitent dépliage et pliage sur les mêmes fonctions.

5.2.3 Pliage/dépliage de systèmes univoques.

Courcelle [Cou79] propose un autre type de condition.

- Un système Σ est *univoque* ssi il a une unique solution dans $M_\Omega^\infty(F,V)/\equiv_R$. Autrement dit, toutes les solutions de Σ, en tant que système d'équations sur les arbres infinis $M_\Omega^\infty(F,V)$, sont R-équivalentes.

L'univocité garantit la correction :

- Proposition [Cou79, th. 5.20] : si Σ **ufld**$_R$ Σ' et Σ' est R-univoque alors $\Sigma \equiv_R \Sigma'$.

Comme le remarque Kott [Kot85], cette approche est à la fois très puissante, car elle assure la correction de toute transformation **ufld**$_R$, et relativement difficile d'emploi, car la vérification de l'univocité fait appel à des propriétés complexes des systèmes de réécriture, comme la confluence et la terminaison.

5.2.4 Dépliage/pliage faible.

C'est ce qui conduit Kott [Kot80, Kot85] vers l'étude de transformations moins générales, mais dont la preuve de correction est moins complexe.

Le système d'équations $\Sigma' = \{\varphi_i(x_{i,1},\ldots,x_{i,n_i}) = t'_i \mid i \in [N]\}$ est obtenu par *dépliage/pliage faible* (weak unfold/fold) de $\Sigma = \{\varphi_i(x_{i,1},\ldots,x_{i,n_i}) = t_i \mid i \in [N]\}$, noté Σ **wuf**$_R$ Σ', ssi :

$$\Sigma\ \mathbf{wuf}_R\ \Sigma' \quad \text{ssi} \quad \text{pour tout } i \in [N] \quad t_i\ (\xrightarrow[\Sigma]{*}\ \xleftrightarrow[R]{*}\ \xleftarrow[\Sigma]{*})\ t'_i$$

Autrement dit, les équations subissent tout d'abord des dépliages, puis des réécritures suivant les lois algébriques, et enfin des pliages.

La transformation **wuf**$_R$ n'est pas toujours correcte car elle comporte des pliages ; c'est un cas particulier de **ufld**$_R$. Nous avons donc la proposition suivante (cf. §5.2.1).

- Proposition : si Σ **wuf**$_R$ Σ' alors $\Sigma \geqslant_R \Sigma'$.

Kott propose d'ajouter un ensemble de règles R' à R afin d'obtenir des conditions pour $\Sigma \equiv_{R\cup R'} \Sigma'$. Il en déduit des conditions suffisantes pour $\Sigma \equiv_R \Sigma'$. Nous renvoyons le lecteur à [Kot80, Kot85] pour l'expression de ces conditions. La section 6.3 de [Kot80] donne des motivations supplémentaires concernant cette approche.

Cette restriction des transformations **ufld**$_R$ est suffisante pour traiter « manuellement » un grand nombre de situations. Cependant, elle est trop restrictive dans le cas d'un système *automatique* de transformation de programmes qui, s'il conclut souvent par des pliages, combine généralement « par nécessité » des réécritures suivant les lois algébriques et des dépliages. Cette transformation a toutefois été spécifiquement employée à des fins d'optimisation dans une procédure automatique pour éliminer des calculs redondants et des parcours de structures inutiles [PP91b].

Néanmoins, Kott signale également que toute transformation **ufld**$_R$ se ramène à une transformation **wuf**$_R$ lorsque les règles de R sont linéaires à gauche *et* à droite [Kot80, prop. 7.3.1]. Une démonstration explicite est aussi donnée dans [Cou86, lemme 1.9]. Cette condition couvre l'essentiel des cas pratiques mais bute par exemple sur la distributivité et, réciproquement, la factorisation $f(x,g(y,z)) \leftrightarrow g(f(x,y),f(x,z))$, ainsi que sur l'idempotence $f(x,x) \to x$, qui intervient dans le partage d'expressions. Nous examinons dans la section suivante un moyen d'étendre la condition de linéarité qui permet de ramener une transformation $\mathbf{ufld}_R$ à une transformation $\mathbf{wuf}_R$.

5.3 Lemmes de commutativité.

Pour ramener une transformation $\mathbf{ufld}_R$ à une transformation $\mathbf{wuf}_R$, on permute les réductions $\to_\Sigma$ et $\to_R$ qui la composent. C'est pourquoi nous examinons les propriétés de commutativité des systèmes de réécriture.

5.3.1 Dérivations et superposition.

Nous affinons ici les définitions de base concernant la réécriture (cf. §N.17), qui ont déjà été exploitées pour la sémantique opérationnelle des schémas de programme (cf. §2.5.2). Pour permettre une manipulation plus abstraite, nous distinguons également $M \to_{u,r,\sigma} N$, réduction *concréte* de M en N à l'occurrence u par la règle r au moyen de la substitution σ, d'une relation $\to_{u,r}$ sur les termes qui exprime simplement la possiblité *abstraite* d'une réduction par r à l'occurrence u.

Définition 5.1. (Réduction)
Soient F une signature et $R \subset M(F,X) \times M(F,X)$ un ensemble des règles. Une *réduction* (ou *dérivation atomique*) *concrète* est un triplet (u, r, σ) tel que :

- $u \in Occ$
- $r = \langle \alpha \to \beta \rangle \in R$
- $\sigma \in Subst(F, X)$

Elle définit une relation $\to_{u,r,\sigma}$ sur $M(F,X)$ par :

- $M \to_{u,r,\sigma} N$ ssi $M/u = \sigma\alpha$ et $N = M[u \leftarrow \sigma\beta]$

Une *réduction abstraite* est un couple (u, r). Elle définit une relation $\to_{u,r}$ sur $M(F,X)$ par :

- $M \to_{u,r} N$ ssi $\exists \sigma \in Subst(F,X) \quad M \to_{u,r,\sigma} N$

Nous définissons sur $M(F,X)$ la relation $\to_R$ de *réduction par* R :

- $M \to_R N$ ssi $\exists u \in Occ \quad \exists r \in R \quad M \to_{u,r} N$

On omet l'indice R lorsqu'il n'est pas ambigu. Règle et réduction réciproques sont notées ainsi :

- $r^{-1} = \langle \beta \to \alpha \rangle$ et $R^{-1} = \{r^{-1} \mid r \in R\}$ et $(u, r, \sigma)^{-1} = (u, r^{-1}, \sigma)$ et $(u, r)^{-1} = (u, r^{-1})$
- $M \leftarrow_{u,r,\sigma} N$ ssi $M \to_{u,r^{-1},\sigma} N$ ssi $N \to_{u,r,\sigma} M$

On a de même $\leftarrow_{u,r} = (\to_{u,r})^{-1} = \to_{u,r^{-1}}$ pour la réduction abstraite. On note $Reduc(R)$ l'ensemble des réductions sur R.

Définition 5.2. (Dérivation parallèle)
Soient F une signature et $R \subset M(F,X) \times M(F,X)$ un ensemble des règles. Une *dérivation parallèle* concrète est une famille $\partial = (u_i, r_i, \sigma_i)_{i \in I}$ de réductions telle que :

- $(u_i)_{i \in I}$ est une famille finie d'occurrences disjointes

Elle définit une relation $\twoheadrightarrow_\partial$ sur $M(F,X)$ par :

- $M \twoheadrightarrow_\partial N$ ssi $\forall i \in I \quad M/u_i = \sigma_i\alpha_i$ et $N = M[u_i \leftarrow \sigma_i\beta_i \mid i \in I]$

On note également :

- $Deriv_{\|}(R)$ l'ensemble des dérivations *parallèles* avec des règles de R
- $Occ(\partial) = \{u_i \mid i \in I\}$
- $Roots(\partial) = Roots(Occ(\partial))$ et $Roots(\partial_1, \ldots, \partial_n) = Roots(Occ(\partial_1) \cup \ldots \cup Occ(\partial_n))$
- $\partial_{|J} = (u_j, r_j, \sigma_j)_{j \in J}$ pour $J \subset I$, et $\partial_{|i} = (u_i, r_i, \sigma_i)$ pour $i \in I$
- $u \cdot \partial = (u \cdot u_i, r_i, \sigma_i)_{i \in I}$ pour toute occurrence $u \in Occ$
- $\sigma.\partial = (u_i, r_i, \sigma.\sigma_i)_{i \in I}$ pour toute substitution $\sigma \in Subst(F, X)$

Si $\partial = (u_i, r_i)_{i \in I}$ et $\partial' = (u_i, r_i)_{i \in I'}$ sont deux dérivations parallèles de $Deriv_{\|}(R)$, indicées par des ensembles I et I' disjoints et telles que $Occ(\partial) \parallel Occ(\partial')$, on note alors :

- $\partial \cup \partial' = (u_i, r_i, \sigma_i)_{i \in I \cup I'} \in Deriv_{\|}(R)$

La *dérivation réciproque* est $\partial^{-1} = (u_i, r_i^{-1}, \sigma_i)_{i \in I}$. On a :

- $\twoheadleftarrow_{\partial} = (\twoheadrightarrow_{\partial})^{-1} = \twoheadrightarrow_{\partial^{-1}} \in Deriv_{\|}(R^{-1})$

En pratique, $U = Occ(\partial)$ lui-même est l'ensemble d'indices. On a alors :

- $\partial = (u, r_u, \sigma_u)_{u \in U}$

On définit la même terminologie sur les *dérivations abstraites parallèles* $\partial = (u_i, r_i)_{i \in I}$. On note en particulier $\twoheadrightarrow_{\partial}$ et $\twoheadrightarrow_R$ les relations sur $M(F, X)$ définies par :

- $M \twoheadrightarrow_{\partial} N$ ssi $\exists (\sigma_i)_{i \in I} \in Subst(F, X)^I \quad M \twoheadrightarrow_{(u_i, r_i, \sigma_i)_{i \in I}} N$
- $M \twoheadrightarrow_R N$ ssi $\exists\, \partial \in Deriv_{\|}(R) \quad M \twoheadrightarrow_{\partial} N$

On omet l'indice R lorsqu'il n'est pas ambigu.

Définition 5.3. (Dérivation)
Soient F une signature et $R \subset M(F, X) \times M(F, X)$ un ensemble des règles. Une *dérivation de longueur* n est une séquence finie de réductions $\nabla = (u_i, r_i, \sigma_i)_{i \in [n]}$. Elle définit une relation $\twoheadrightarrow_{\nabla}^{n}$ sur $M(F, X)$ par :

- $M_0 \rightarrow_{\nabla}^{n} M_n$ ssi $\exists (M_i)_{i \in [n]} \in M(F, X)^n \quad \forall i \in [n] \quad M_{i-1} \rightarrow_{u_i, r_i, \sigma_i} M_i$

On note également :

- $Roots(\nabla) = Roots(u_1, \ldots, u_n)$
- $|\nabla| = n$
- $\nabla^{-1} = (u_{n-i}, r_{n-i}^{-1}, \sigma_{n-i})_{i \in [n]}$
- $u \cdot \nabla = (u \cdot u_i, r_i, \sigma_i)_{i \in [n]}$ pour toute occurrence $u \in Occ$
- $\sigma.\nabla = (u_i, r_i, \sigma.\sigma_i)_{i \in [n]}$ pour toute substitution $\sigma \in Subst(F, X)$
- $Deriv(R) = Reduc(R)^*$ est l'ensemble des dérivations qui emploient des règles de R

On définit la même terminologie sur les *dérivations abstraites* $\nabla = (u_i, r_i)_{i \in [n]}$. On note en particulier $\rightarrow_{\nabla}^{n}$ et $\rightarrow^n$ les relations sur $M(F, X)$ définies par :

- $M \rightarrow_{\nabla}^{n} N$ ssi $\exists (\sigma_i)_{i \in [n]} \in Subst(F, X)^{[n]} \quad M \twoheadrightarrow_{(u_i, r_i, \sigma_i)_{i \in I}} N$
- $M \rightarrow^n N$ ssi $\exists\, \nabla \in Reduc(R)^n \quad M \rightarrow_{\nabla}^{n} N$

Les relations $\rightarrow_{\nabla}^{*}$ pour $\nabla \in Deriv(R)$, et $\rightarrow_{R}^{*}$, ne font pas mention explicite de la longueur de dérivation :

- $M \rightarrow_{\nabla}^{*} N$ ssi $\exists (\sigma_i)_{i \in [|\nabla|]} \in Subst(F, X)^{[|\nabla|]} \quad M$
- $M \rightarrow_{R}^{*} N$ ssi $\exists \nabla \in Deriv \quad (R) M \rightarrow_{\nabla}^{*} N$

La donnée d'une séquence $\nabla = (\partial_i)_{i \in [n]}$ de dérivations parallèles définit également une relation de dérivation $\rightarrow_{\nabla}^{*}$ qui consiste en un ordre arbitraire des réductions de chacune des dérivations parallèles :

- $M_0 \rightarrow_{\nabla}^{n} M_n$ ssi $\exists (M_i)_{i \in [n]} \in M(F, X)^n \quad \forall i \in [n] \quad M_{i-1} \twoheadrightarrow_{\partial_i} M_i$

Soit $\nabla = (\partial_i)_{i\in[n]}$ et $\nabla' = (\partial_{i+n})_{i\in[n']}$. On note :

- $Roots(\nabla) = Roots(\partial_1, \ldots, \partial_n)$
- $\nabla^{-1} = (\partial_{n-i}^{-1})_{i\in[n]}$.
- $\nabla; \nabla' = (\partial_i)_{i\in[n+n']}$
- si $Roots(\nabla) \parallel Roots(\nabla')$ alors $\rightarrow^*_{\nabla;\nabla'} = \rightarrow^*_{\nabla} \rightarrow^*_{\nabla'} = \rightarrow^*_{\nabla'} \rightarrow^*_{\nabla}$

Plus généralement, si $(\nabla_i)_{i\in[m]}$ est une famille de dérivations,

- $\nabla = \prod_{i\in[m]} \nabla_i = \nabla_1; \ldots; \nabla_m$

La relation ∇ résultante est indépendante de l'ordre des ∇_i si les $Roots(\nabla_i)$ sont parallèles deux à deux. Autrement dit, les relations $\rightarrow_{\nabla_i}$ permutent entre elles. Il n'est pas indispensable alors que l'ensemble d'incides soit explicitement ordonné.

Les problèmes de confluence et de commutativité font intervenir les notions de superposition et de paire critique [KB70, Hue80].

Définition 5.4. (Superposition)
Un terme $M_2 \in M(F, X)$ *se superpose* (overlap) à un terme $M_1 \in M(F, X)$ à l'occurrence $u \in Occ(M_1)$ ssi $M_1(u) \notin Var(M_1)$ et $\{M_1/u, M_2\}$ est unifiable. Une règle $r_2 = \langle \alpha_2 \rightarrow \beta_2 \rangle$ *se superpose à* une règle $r_1 = \langle \alpha_1 \rightarrow \beta_1 \rangle$ à l'occurrence u ssi α_2 se superpose à α_1 en u. On parle alors de *superposition de r_2 sur r_1 en u*. La superposition est *impropre* lorsque $\alpha_2 \in X$ et $u = \epsilon$, et *propre* dans le cas contraire.

Définition 5.5. (Paire critique)
Soient $r_1 = \langle \alpha_1 \rightarrow \beta_1 \rangle$ et $r_2 = \langle \alpha_2 \rightarrow \beta_2 \rangle$ deux règles, éventuellement renommées afin qu'elles n'aient pas de variables communes, et $u \in Occ(\alpha_1)$ tel que r_2 se superpose à r_1 en u. Si $r_1 = r_2$, on suppose en outre que $u \neq \epsilon$. Soit σ un unificateur principal de α_1/u et α_2. On dit que la superposition de r_2 sur r_1 en u détermine une *paire critique* $\langle P, Q \rangle$ définie par $P = \sigma(\alpha_1[u \leftarrow \beta_2])$ et $Q = \sigma(\beta_1)$. Une paire critique $\langle P, Q \rangle$ est unique à un renommage près. Elle est *propre* (resp. *impropre*) ssi la superposition est propre (resp. impropre).

La définition ordinaire de la superposition considère qu'une règle r_2 de la forme $\langle x \rightarrow \beta_2 \rangle$ se superpose à toute règle $r_1 = \langle \alpha_1 \rightarrow \beta_1 \rangle$ en ϵ. Cette superposition est très particulière car elle ne met pas de constructeurs de r_1 et r_2 en commun. Intuitivement, la situation est identique au cas où r_1 apparaît sous l'occurrence d'une variable de r_2 C'est pourquoi nous avons distingué superposition *propre* et *impropre*.

Pour la commutativité, nous nous appuyons sur Toyama [Toy88], qui définit une notion de paire critique entre deux systèmes de règles.

Définition 5.6. (Ensemble des paires critiques)
Soient r_1 et r_2 deux règles. On note $Crit(r_1, r_2)$ l'ensemble des paires critiques déterminées par des superpositions de r_2 sur r_1. Soient R_1 et R_2 deux ensembles de règles, on note $Crit(R_1, R_2) = \bigcup_{r_1\in R_1, r_2\in R_2} Crit(r_1, r_2)$. L'*ensemble des paires critiques* d'un système R est $Crit(R) = Crit(R, R)$. On note de même $Crit_{\text{pro}}(R_1, R_2)$ et $Crit_{\text{pro}}(R)$ les ensembles de paires critiques propres.

Il faut noter que la définition de la superposition est asymétrique. En général on a donc $Crit(r_1, r_2) \neq Crit(r_2, r_1)$. Par définition, nous avons aussi $Crit_{\text{pro}}(R_1, R_2) \subset Crit(R_1, R_2)$.

5.3.2 Commutativité et linéarité à gauche.

Nous nous intéressons dans cette section à la commutativité des systèmes de réécriture linéaires à gauche. Les résultats qu'elle contient apparaissent sous diverses formes ou variantes dans [Ros73, th. 6.5], [RV80, prop. 10], [Hue80, lemme 3.3] et [Toy88, cor. 3.1].

Il faut noter que nos propositions sont à la fois plus particulières, parce que nous ne considérons que des règles sans superposition (nous autorisons toutefois les superpositions impropres), et plus générales, parce que nous n'imposons pas que les règles $\langle \alpha \rightarrow \beta \rangle$ vérifient $Var(\alpha) \supset Var(\beta)$. Cette condition supplémentaire, qui rend la réduction déterministe, n'est pas vérifiée par les règles de $R(\boldsymbol{D})$ ou $V(\boldsymbol{D})$ (cf. §2.5.2). Il ne nous est donc pas possible d'utiliser directement des résultats connus. Les propositions voisines données à la sous-section suivante (§5.3.3), plus originales, n'emploient pas seulement la linéarité à gauche mais aussi la confluence.

Lemme 5.1. (Linéarité à gauche et superposition)
Soit $\partial = (u, r_u)_{u \in U}$ une dérivation parallèle par un ensemble de règles R et $\partial_0 = (u_0, r_0)$ une réduction. Supposons vérifiées les hypothèses suivantes :

- $u_0 \leqslant U$
- $\forall u \in U \quad r_u$ ne se superpose pas à r_0 en u/u_0
- r_0 est linéaire à gauche

Il existe alors une dérivation parallèle $\partial' \in Deriv_{\parallel}(R)$ de domaine U' dominé par u_0, telle que :

$$\underset{\partial_0}{\longleftarrow}\ \underset{\partial}{\overset{\parallel}{\twoheadrightarrow}} \quad \subset \quad \underset{\partial'}{\overset{\parallel}{\twoheadrightarrow}}\ \underset{\partial_0}{\longleftarrow}$$

De plus, si r_0 est linéaire à droite, alors $|U'| \leqslant |U|$. Si r_0 est d'ordre zéro à droite, alors $U' = \varnothing$.

Démonstration. La démonstration formelle de ce lemme est donnée en annexe (cf. §D.5). Elle suit les lignes de celle du lemme 3.3 de Huet [Hue80].

Soit une dérivation $M \leftarrow_{u_0, r_0, \sigma_0} N \twoheadrightarrow_{(u, r_u, \sigma_u)_{u \in U}} P$ avec $r_0 = \langle \alpha_0 \rightarrow \beta_0 \rangle$ et $r_u = \langle \alpha_u \rightarrow \beta_u \rangle$ pour $u \in U$. Par définition, cela signifie $N/u_0 = \sigma_0 \alpha_0$, $M = N[u_0 \leftarrow \sigma_0 \beta_0]$, et pour tout $u \in U$, $N/u = \sigma_u \alpha_u$, et $P = N[u \leftarrow \sigma_u \beta_u]$.

Pour tout $u \in U$, $u_0 \leqslant u$ et aucun r_u ne se superpose à r_0 en u/u_0 signifie que la reduction en u a lieu sous une variable x_u de $Var(\alpha_0)$ dans le terme $N/u_0 = \sigma_0 \alpha_0$.

Pour tout $x \in Var(\alpha_0)$ fixé, les exemplaires de $x_u = x$ sous β_0 dans M, comme ceux situés sous α_0 dans N, sont tous instanciés par le même terme $\sigma_0 x$. Puisque α_0 est linéaire, les occurrences de $\sigma_u \alpha_u$ sous x dans $\sigma_0 x$ sont disjointes. On peut donc réduire en parallèle les termes $\sigma_u \alpha_u$ et construire une dérivation $M \twoheadrightarrow_{\partial'} M'$.

Puisque l'on a fait la même opération sous chaque occurrence des x de β_0 dans M, il revient au même de réduire d'abord $N \twoheadrightarrow_{\partial} P$ puis $P \twoheadrightarrow_{\partial_0} M'$. Ces dérivations sont illustrées à la figure 5.1. $\square$

Lemme 5.2. (Linéarité à gauche et dérivations parallèles)
Soient R_1, R_2 des ensembles de règles linéaires à gauche tels que $Crit_{\text{pro}}(R_1, R_2) = Crit_{\text{pro}}(R_2, R_1) = \varnothing$.

$$\underset{R_1}{\overset{\parallel}{\twoheadleftarrow}}\ \underset{R_2}{\overset{\parallel}{\twoheadrightarrow}} \quad \subset \quad \underset{R_2}{\overset{\parallel}{\twoheadrightarrow}}\ \underset{R_1}{\overset{\parallel}{\twoheadleftarrow}}$$

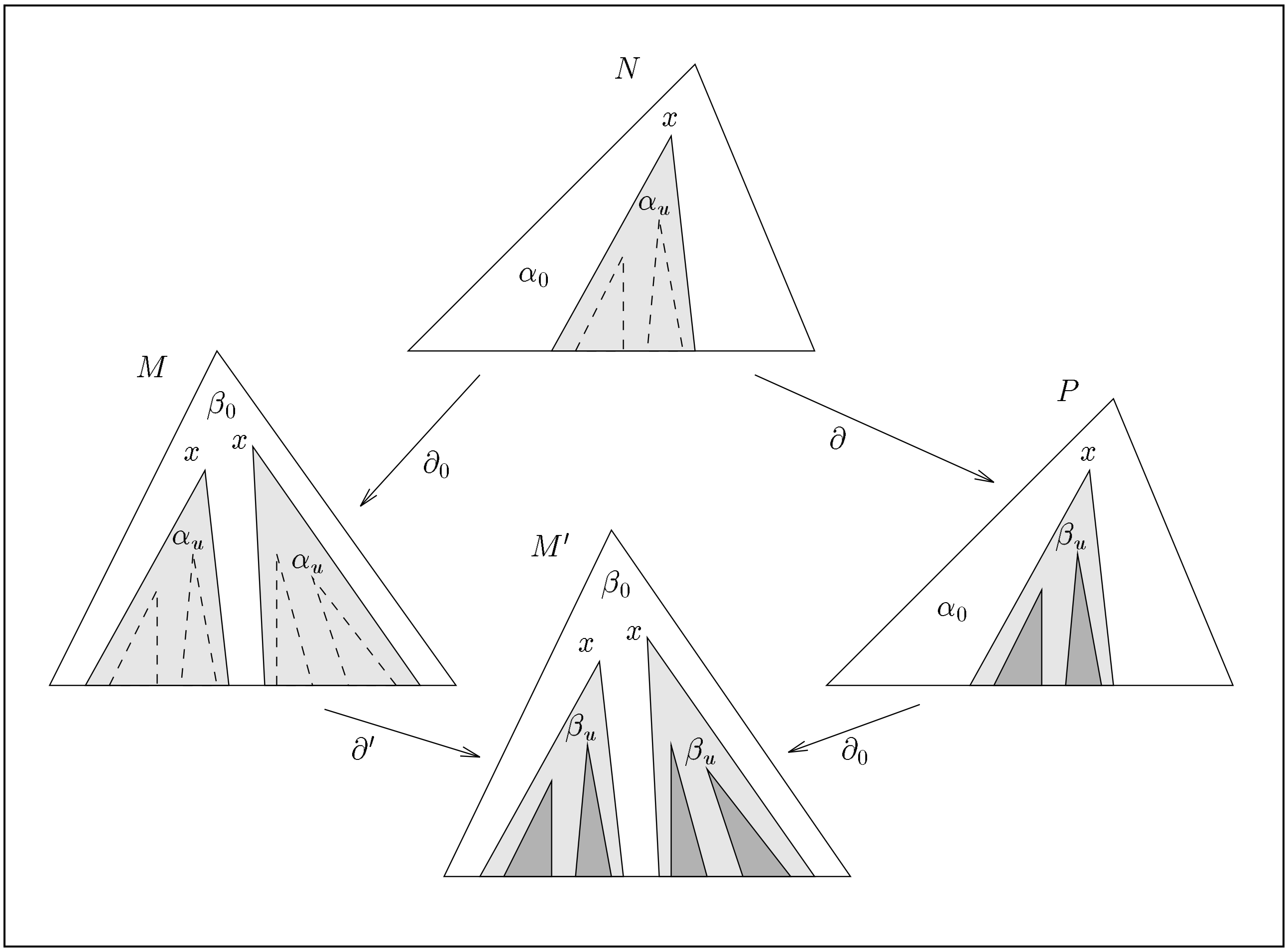

Figure 5.1 : Commutativité et linéarité à gauche

Démonstration. La démonstration formelle de ce lemme est donnée en annexe (cf. §D.5).

Soient $\partial_1 \in \mathit{Deriv}_{\parallel}(R_1)$ et $\partial_2 \in \mathit{Deriv}_{\parallel}(R_2)$ deux dérivations parallèles de domaine U_1 et U_2. Il y a quatre possibilités à considérer lorsqu'on compose $\overset{\shortparallel}{\leftarrow}_{\partial_1}$ et $\overset{\shortparallel}{\rightarrow}_{\partial_2}$.

1. Si $u_1 \in U_1$ domine strictement $V_2 = \{u_2 \in U_2 \mid u_1 < u_2\} \neq \varnothing$, alors $\leftarrow_{\partial_1|u_1} \overset{\shortparallel}{\rightarrow}_{\partial_2|V_2} \subset \overset{\shortparallel}{\rightarrow}_{\partial'_2} \leftarrow_{\partial_1|u_1}$ par le lemme 5.1.
2. Si $u_2 \in U_2$ domine strictement $V_1 = \{u_1 \in U_1 \mid u_2 < u_1\} \neq \varnothing$, alors $\overset{\shortparallel}{\leftarrow}_{\partial_1|V_1} \rightarrow_{\partial_2|u_2} \subset \rightarrow_{\partial_2|u_2} \overset{\shortparallel}{\leftarrow}_{\partial'_1}$ par le lemme 5.1, symétriquement.
3. S'il existe $u \in U_1 \cap U_2$, notons r_1 et r_2 les régles qui correspondent à u respectivement dans ∂_1 et ∂_2. On a alors deux cas exclusifs « r_2 ne se superpose pas à r_1 en ϵ » et « r_1 ne se superpose pas à r_2 en ϵ ». Grâce au lemme 5.1, on obtient $\leftarrow_{\partial_1|u_1} \rightarrow_{\partial_2|u_2} \subset \overset{\shortparallel}{\rightarrow}_{\partial'_2} \leftarrow_{\partial_1|u_1}$ dans le premier cas, et $\leftarrow_{\partial_1|u_1} \rightarrow_{\partial_2|u_2} \subset \rightarrow_{\partial_2|u_2} \overset{\shortparallel}{\leftarrow}_{\partial'_1}$ dans le second.
4. Enfin si $u_1 \parallel U_2$ pour $u_1 \in U_1$, la réduction permute avec toutes les autres : $\leftarrow_{\partial_1|u_1} \overset{\shortparallel}{\rightarrow}_{\partial_2} = \overset{\shortparallel}{\rightarrow}_{\partial_2} \leftarrow_{\partial_1|u_1}$. Le cas est identique lorsque, symétriquement, $u_2 \parallel U_1$.

Dans la mesure où U_1 et U_2 sont constitués d'occurrences disjointes, ces cas individuels sont indépendants et peuvent être traités en parallèle. Lorsqu'on les réunit tous, on obtient $\overset{\shortparallel}{\leftarrow}_{\partial_1} \overset{\shortparallel}{\rightarrow}_{\partial_2} \subset \overset{\shortparallel}{\rightarrow}_{\partial'_2} \overset{\shortparallel}{\leftarrow}_{\partial'_1}$. $\square$

Proposition 5.3. (Linéarité à gauche et commutativité)
Soient R_1, R_2 des ensembles de règles linéaires à gauche tels que $Crit_{\mathrm{pro}}(R_1, R_2) = Crit_{\mathrm{pro}}(R_2, R_1) = \varnothing$.

$$\xleftarrow[R_1]{*} \; \xrightarrow[R_2]{*} \quad \subset \quad (\xleftarrow[R_1]{} \cup \xrightarrow[R_2]{})^* \quad = \quad \xrightarrow[R_2]{*} \; \xleftarrow[R_1]{*}$$

Plus précisément, on a $\forall m, n \in \mathbb{N} \quad ⇺^m_{R_1} \; ⇻^n_{R_2} \subset ⇻^n_{R_2} \; ⇺^m_{R_1}$.

Démonstration. On est dans les conditions d'application du lemme 5.2. On a donc $⇺_1 \; ⇻_2 \subset ⇻_2 \; ⇺_1$. On en déduit $⇺^m_1 \; ⇻^n_2 \subset ⇻^n_2 \; ⇺^m_1$ par récurrence successive sur $m, n \in \mathbb{N}$. Par conséquent, $\leftarrow^*_1 \; \rightarrow^*_2 \subset \rightarrow^*_2 \; \leftarrow^*_1$.

On montre ensuite par récurrence sur $n \in \mathbb{N}$ que $(\leftarrow_1 \cup \rightarrow_2)^n \subset \rightarrow^*_2 \; \leftarrow^*_1$. L'identité est triviale pour $n = 0$. Pour le cas strictement positif, on a $(\leftarrow_1 \cup \rightarrow_2)^{n+1} = (\leftarrow_1 \cup \rightarrow_2)^n \; (\leftarrow_1 \cup \rightarrow_2) \subset \rightarrow^*_2 \; \leftarrow^*_1 \; (\leftarrow_1 \cup \rightarrow_2)$ par l'hypothèse de récurrence. La relation se distribue sur les deux cas suivants :

1. $\rightarrow^*_2 \; \leftarrow^*_1 \; \leftarrow_1 \subset \rightarrow^*_2 \; \leftarrow^*_1$.
2. $\rightarrow^*_2 \; \leftarrow^*_1 \; \rightarrow_2 \subset \rightarrow^*_2 \; ⇻_2 \; \leftarrow^*_1$ en vertu du résultat précédent, $\subset \rightarrow^*_2 \; \leftarrow^*_1$.

On a donc bien $(\leftarrow_1 \cup \rightarrow_2)^{n+1} \subset \rightarrow^*_2 \; \leftarrow^*_1$. L'inclusion réciproque $(\leftarrow_1 \cup \rightarrow_2)^* \supset \rightarrow^*_2 \; \leftarrow^*_1$ est triviale. $\square$

Corollaire 5.4. (Linéarité à gauche et confluence)
Pour tout ensemble R de règles linéaires à gauche tel que $Crit_{\mathrm{pro}}(R) = \varnothing$, la relation $\rightarrow_R$ est confluente et

$$\overset{*}{\underset{R}{\longleftrightarrow}} \quad = \quad \xrightarrow[R]{*} \; \xleftarrow[R]{*}$$

5.3.3 Commutativité et confluence.

Nous adaptons les résultats de la section précédente (§5.3.2) au cas où un seul des deux systèmes est linéraire à gauche. On exploite alors sa confluence (lemme 5.4). Il n'y a pas strictement commutativité dans ce cas ; il subsiste un « résidu de dérivation ».

Lemme 5.5. (Confluence d'une famille de dérivations)
Soient R un ensemble de règles qui détermine une relation $\rightarrow_R$ confluente, $(\nabla_i)_{i \in I}$ une famille finie de dérivations, et $x, (y_i)_{i \in I} \in M(F, X)$.

- si $\forall i \in I \quad x \rightarrow^*_{\nabla_i} y_i$ alors $\exists x' \in M(F, X) \quad \exists (\nabla'_i)_{i \in I} \in Deriv(R)^I \quad \forall i \in I \quad y_i \rightarrow^*_{\nabla'_i} x'$

Démonstration. On montre ce lemme par récurrence sur le nombre d'éléments de I. Il est trivial si I est vide ou un singleton. Si I contient au moins deux éléments j et k, par l'hypothèse de récurrence, il existe $x'' \in M(F, X)$ et $(\nabla''_i)_{i \in I \setminus \{j\}} \in Deriv(R)^{I \setminus \{j\}}$ tels que $\forall i \in I \setminus \{j\} \;\; y_i \rightarrow^*_{\nabla'_i} x''$. Puisque $\rightarrow_R$ est confluente et $y_j \leftarrow^*_{\nabla_j} x \rightarrow^*_{\nabla''_k} x''$, il existe ∇'_j, ∇' et x' tels que $y_j \rightarrow^*_{\nabla'_j} x' \leftarrow^*_{\nabla'} x''$. Pour tout $i \in I \setminus \{j\}$, on définit $\nabla'_i = \nabla''_i ; \nabla'$. On a alors $\forall i \in I \;\; y_i \rightarrow^*_{\nabla'_i} x'$. $\square$

Lemme 5.6. (Confluence et superposition)
Soit $\partial = (u, r_u)_{u \in U}$ une dérivation parallèle par un ensemble de règles R et $\partial_0 = (u_0, r_0)$ une réduction. Supposons vérifiées les hypothèses suivantes :

- $u_0 \leqslant U$
- $\forall u \in U \quad r_u$ ne se superpose pas à r_0 en u/u_0
- R est confluent

Il existe alors deux dérivations ∇ et ∇' de R, dominées par u_0 et telles que :

$$\underset{\partial_0}{\longleftarrow}\ \underset{\partial}{\twoheadrightarrow}\ \subset\ \underset{\nabla}{\overset{*}{\longrightarrow}}\ \underset{\partial_0}{\longleftarrow}\ \underset{\nabla'}{\overset{*}{\longleftarrow}}$$

Si β_0 est d'ordre zéro à droite, alors $\nabla = \varnothing$. Le cas où r_0 est linéaire à gauche est traité plus en détail par le lemme 5.1 (que R soit confluent est alors sans importance).

Démonstration. La démonstration formelle de ce lemme est donnée en annexe (cf. §D.5).

Soit une dérivation $M \longleftarrow_{\partial_0} N \twoheadrightarrow_{\partial} P$ avec $\partial_0 = (u_0, \langle \alpha_0 \rightarrow \beta_0 \rangle)$. Comme pour le lemme 5.1, la condition de superposition signifie que la réduction en $u \in U = Occ(\partial)$ a lieu sous une variable x_u de $Var(\alpha_0)$ dans le terme $N/u_0 = \sigma_0 \alpha_0$.

Pour tout $x \in Var(\alpha_0)$ fixé, les exemplaires de $x_u = x$ sous β_0 dans M, comme ceux situés sous α_0 dans N, sont tous instanciés par le même terme $\sigma_0 x$. Dans la mesure où α_0 n'est pas nécessairement linéaire, ces occurrences, disjointes dans N, peuvent se recouvrir dans M après transport par ∂_0 (voir la figure 5.2).

Toutefois, l'hypothèse de confluence permet de prolonger grâce au lemme 5.5 les réductions aux occurrences u en des dérivations qui confluent vers un même terme pour x fixé, et cela soit en partant de M : $M \rightarrow^*_{\nabla} M'$, soit en partant de P : $P \rightarrow^*_{\nabla'} N'$. Le fait que les variables $x \in Var(\alpha_0)$ soient chacunes instanciées par un même terme permet de construire la substitution qui justifie la réduction $M' \longleftarrow_{\partial_0} N'$. Ces dérivations sont illustrées à la figure 5.2. □

Lemme 5.7. (Linéarité à gauche et dérivations parallèles)
Soient R_1, R_2 des ensembles de règles tels que R_1 est linéaire à gauche et $Crit_{\text{pro}}(R_1, R_2) = Crit_{\text{pro}}(R_2, R_1) = Crit_{\text{pro}}(R_1) = \varnothing$, alors :

$$\underset{R_1}{\twoheadleftarrow}\ \underset{R_2}{\twoheadrightarrow}\ \subset\ \underset{R_1}{\overset{*}{\longrightarrow}}\ \underset{R_2}{\twoheadrightarrow}\ \underset{R_1}{\overset{*}{\longleftarrow}}$$

Démonstration. La démonstration formelle de ce lemme est à l'image de celle donnée en annexe (cf. §D.5) pour le lemme 5.2, par récurrence sur la somme des longueurs des dérivations parallèles. Nous ne donnons ici que les cas individuels.

Soient $\partial_1 \in Deriv_{\|}(R_1)$ et $\partial_2 \in Deriv_{\|}(R_2)$ deux dérivations parallèles de domaine U_1 et U_2. Il y a quatre possibilités à considérer lorsqu'on compose $\twoheadleftarrow_{\partial_1}$ et $\twoheadrightarrow_{\partial_2}$.

1. Si $u_1 \in U_1$ domine strictement $V_2 = \{u_2 \in U_2 | u_1 < u_2\} \neq \varnothing$, alors $\longleftarrow_{\partial_1|u_1} \twoheadrightarrow_{\partial_2|V_2} \subset \twoheadrightarrow_{\partial_2'} \longleftarrow_{\partial_1|u_1}$ par le lemme 5.1.
2. Si $u_2 \in U_2$ domine strictement $V_1 = \{u_1 \in U_1 | u_2 < u_1\} \neq \varnothing$, $\twoheadleftarrow_{\partial_1|V_1} \longrightarrow_{\partial_2|u_2} \subset \longrightarrow^*_{\nabla_1} \longrightarrow_{\partial_2|u_2} \longleftarrow^*_{\nabla_1'}$ grâce au lemme 5.6 car R_1 est confluent (lemme 5.4).

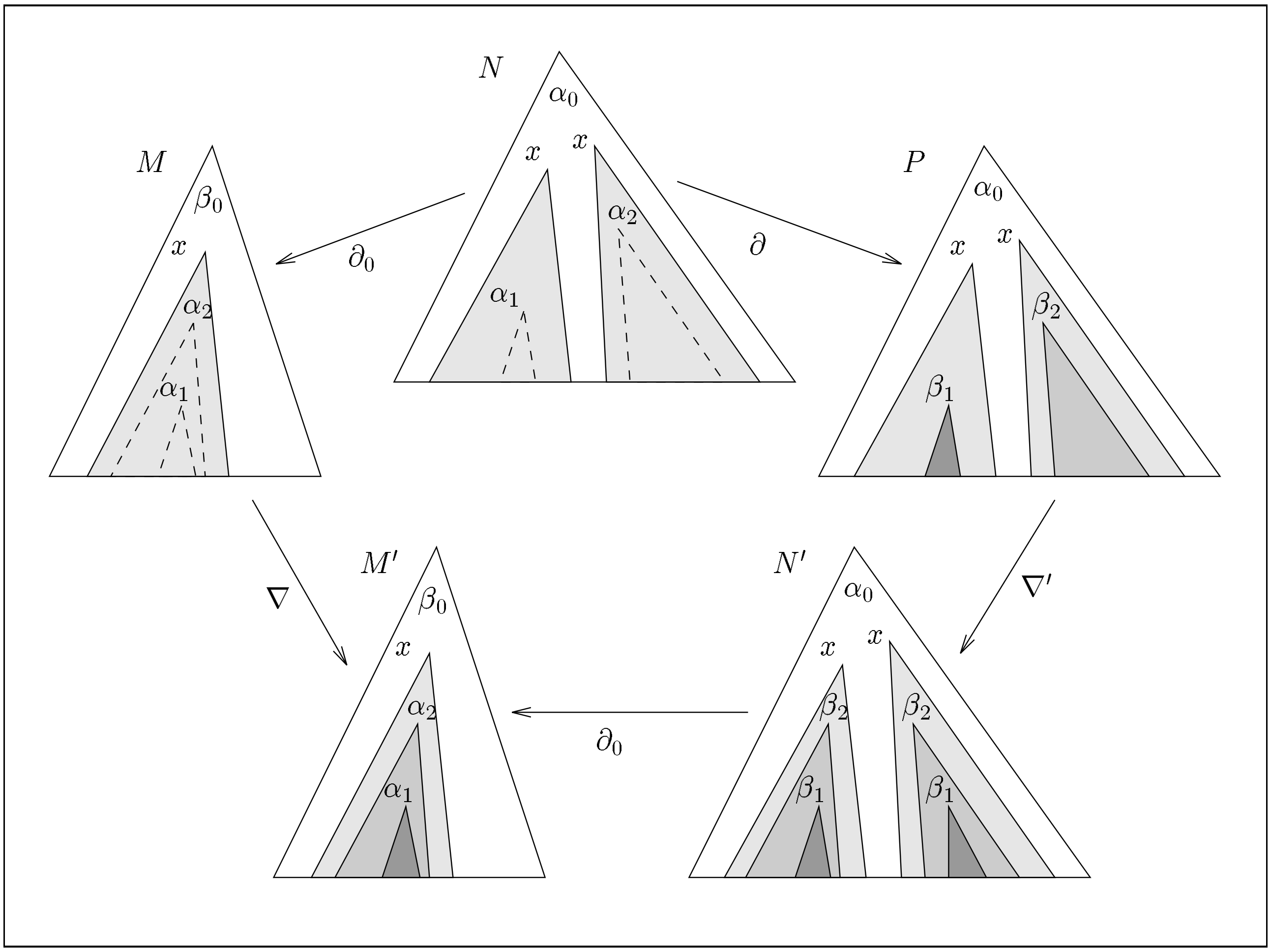

Figure 5.2 : Commutativité et confluence

3. S'il existe $u \in U_1 \cap U_2$, notons r_1 et r_2 les régles qui correspondent à u respectivement dans ∂_1 et ∂_2. On a alors deux cas exclusifs « r_2 ne se superpose pas à r_1 en ϵ » et « r_1 ne se superpose pas à r_2 en ϵ ». On obtient alors $\leftarrow_{\partial_1|u_1} \rightarrow_{\partial_2|u_2} \subset \twoheadrightarrow_{\partial'_2} \leftarrow_{\partial_1|u_1}$ grâce au lemme 5.1 dans le premier cas, et $\leftarrow_{\partial_1|u_1} \rightarrow_{\partial_2|u_2} \subset \rightarrow^*_{\nabla_1} \rightarrow_{\partial_2|u_2} \leftarrow^*_{\nabla'_1}$ par le lemme 5.6 dans le second.
4. Enfin si $u_1 \parallel U_2$ pour $u_1 \in U_1$, la réduction permute avec toutes les autres : $\leftarrow_{\partial_1|u_1} \twoheadrightarrow_{\partial_2} = \twoheadrightarrow_{\partial_2} \leftarrow_{\partial_1|u_1}$. Le cas est identique lorsque, symétriquement, $u_2 \parallel U_1$.

Dans la mesure où U_1 et U_2 sont constitués d'occurrences disjointes, ces cas individuels sont indépendants et peuvent être traités en parallèle. Lorsqu'on les réunit tous, on obtient $\twoheadleftarrow_{\partial_1} \twoheadrightarrow_{\partial_2} \subset \rightarrow^*_{\nabla_1} \twoheadrightarrow_{\partial_2} \leftarrow^*_{\nabla'_1}$. $\square$

5.3.4 Retour sur la linéarité à gauche.

On regroupe les résultats obtenus aux deux sections prédécentes (cf. §5.3.2, 5.3.3) dans le but de simplifier l'expression de la relation $\xleftrightarrow[R_1,R_2]{*}$. On n'obtient en fait de résultat que sur $(\xleftrightarrow[R_1]{} \cup \xrightarrow[R_2]{})^*$ grâce au théorème 5.10.

Lemme 5.8. (Linéarité à gauche et dérivations parallèles)
Soient R_1, R_2 des ensembles de règles tels que R_1 est linéaire à gauche et $Crit_{\text{pro}}(R_1, R_2) = Crit_{\text{pro}}(R_2, R_1) =$

$Crit_{\mathrm{pro}}(R_1) = \varnothing$, alors :

$$\xleftarrow[R_1]{*} \; \xrightarrow[R_2]{\shortparallel} \quad \subset \quad \xrightarrow[R_1]{*} \; \xrightarrow[R_2]{\shortparallel} \; \xleftarrow[R_1]{*}$$

Démonstration. On montre par récurrence sur $n \in N$ que $\mathrel{\leftarrow\!\!\!\!\shortparallel}_1^n \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \subset \rightarrow_1^* \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \leftarrow_1^*$. L'identité est triviale pour $n = 0$. Pour le cas strictement positif, on a $\mathrel{\leftarrow\!\!\!\!\shortparallel}_1^{n+1} \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 = \mathrel{\leftarrow\!\!\!\!\shortparallel}_1 \mathrel{\leftarrow\!\!\!\!\shortparallel}_1^n \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \subset \mathrel{\leftarrow\!\!\!\!\shortparallel}_1 \rightarrow_1^* \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \leftarrow_1^*$ par l'hypothèse de récurrence, $\subset \rightarrow_1^* \mathrel{\leftarrow\!\!\!\!\shortparallel}_1 \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \leftarrow_1^*$ par la proposition 5.3, $\subset \rightarrow_1^* \rightarrow_1^* \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \leftarrow_1^* \leftarrow_1^*$ par le lemme 5.7, $\subset \rightarrow_1^* \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \leftarrow_1^*$. $\square$

Lemme 5.9. (Linéarité à gauche et commutativité)
Soient R_1, R_2 des ensembles de règles linéaires à gauche tels que $Crit_{\mathrm{pro}}(R_1, R_2 \cup R_2^{-1}) = Crit_{\mathrm{pro}}(R_2 \cup R_2^{-1}, R_1) = Crit_{\mathrm{pro}}(R_1) = \varnothing$, alors :

$$\xrightarrow[R_2]{*} \; \xrightarrow[R_1]{*} \quad \subset \quad \xrightarrow[R_1]{*} \; \xrightarrow[R_2]{*} \; \xleftarrow[R_1]{*}$$

Démonstration. On montre par récurrence sur $n \in N$ que $\leftarrow_1^* \mathrel{\leftarrow\!\!\!\!\shortparallel}_2^n \subset \rightarrow_1^* \leftarrow_2^* \leftarrow_1^*$. L'identité est triviale pour $n = 0$. Pour le cas strictement positif, on a $\leftarrow_1^* \mathrel{\leftarrow\!\!\!\!\shortparallel}_2^{n+1} = \leftarrow_1^* \mathrel{\leftarrow\!\!\!\!\shortparallel}_2^n \mathrel{\leftarrow\!\!\!\!\shortparallel}_2 \subset \rightarrow_1^* \leftarrow_2^* \leftarrow_1^* \mathrel{\leftarrow\!\!\!\!\shortparallel}_2$ par l'hypothèse de récurrence, $\subset \rightarrow_1^* \leftarrow_2^* \rightarrow_1^* \mathrel{\leftarrow\!\!\!\!\shortparallel}_2 \leftarrow_1^*$ par le lemme 5.8 appliqué à R_2^{-1}, $\subset \rightarrow_1^* \rightarrow_1^* \leftarrow_2^* \mathrel{\leftarrow\!\!\!\!\shortparallel}_2 \leftarrow_1^*$ par par le lemme 5.3, $\subset \rightarrow_1^* \leftarrow_2^* \leftarrow_1^*$. $\square$

Théorème 5.10. (Linéarité à gauche et commutativité)
Soient R_1, R_2 des ensembles de règles linéaires à gauche tels que $Crit_{\mathrm{pro}}(R_1, R_2 \cup R_2^{-1}) = Crit_{\mathrm{pro}}(R_2 \cup R_2^{-1}, R_1) = Crit_{\mathrm{pro}}(R_1) = \varnothing$, alors :

$$(\xleftrightarrow[R_1]{} \cup \xrightarrow[R_2]{})^* \quad = \quad \xrightarrow[R_1]{*} \; \xrightarrow[R_2]{*} \; \xleftarrow[R_1]{*}$$

Démonstration. On montre par récurrence sur $n \in N$ que $(\rightarrow_1 \cup \leftarrow_1 \cup \rightarrow_2)^n \subset \rightarrow_1^* \rightarrow_2^* \leftarrow_1^*$. L'identité est triviale pour $n = 0$. Pour le cas strictement positif, on a $(\rightarrow_1 \cup \leftarrow_1 \cup \rightarrow_2)^{n+1} = (\rightarrow_1 \cup \leftarrow_1 \cup \rightarrow_2)^n \, (\rightarrow_1 \cup \leftarrow_1 \cup \rightarrow_2) \subset (\rightarrow_1^* \rightarrow_2^* \leftarrow_1^*) \, (\rightarrow_1 \cup \leftarrow_1 \cup \rightarrow_2)$ par l'hypothèse de récurrence. On envisage successivement les trois cas.

1. $\rightarrow_1^* \rightarrow_2^* \leftarrow_1^* \rightarrow_1 \subset \rightarrow_1^* \rightarrow_2^* \mathrel{\shortparallel\!\!\!\!\rightarrow}_1 \leftarrow_1^*$ par le lemme 5.3, $\subset \rightarrow_1^* \rightarrow_1^* \rightarrow_2^* \leftarrow_1^* \leftarrow_1^*$ par le lemme 5.9, $\subset \rightarrow_1^* \rightarrow_2^* \leftarrow_1^*$
2. $\rightarrow_1^* \rightarrow_2^* \leftarrow_1^* \rightarrow_2 \subset \rightarrow_1^* \rightarrow_2^* \mathrel{\shortparallel\!\!\!\!\rightarrow}_2 \leftarrow_1^*$ par le lemme 5.3, $\subset \rightarrow_1^* \rightarrow_2^* \leftarrow_1^*$
3. $\rightarrow_1^* \rightarrow_2^* \leftarrow_1^* \leftarrow_1 \subset \rightarrow_1^* \rightarrow_2^* \leftarrow_1^*$

Dans chaque cas, on a bien $(\rightarrow_1 \cup \leftarrow_1 \cup \rightarrow_2)^{n+1} \subset \rightarrow_1^* \rightarrow_2^* \leftarrow_1^*$. L'inclusion réciproque $(\rightarrow_1 \cup \leftarrow_1 \cup \rightarrow_2)^* \supset \rightarrow_1^* \rightarrow_2^* \leftarrow_1^*$ est triviale. $\square$

Notons qu'en substituant R^{-1} à R, on obtient toute une gamme de résultats similaires si l'un ou les deux systèmes R_1, R_2 est linéaire à droite plutôt qu'à gauche.

5.4 Applications aux transformations de pliage/dépliage.

Le système de réécriture $\langle \varphi_i(x_1, \ldots, x_{n_i}) \to t_i \rangle_{i \in [N]}$ associé à un système d'équations Σ est très particulier. D'une part, pour tout $i \in [N]$, les variables $(x_j)_{j \in [n_i]}$ sont distinctes. Par conséquent,

$$\Sigma \text{ est linéaire à gauche}$$

D'autre part, les inconnues $(\varphi_i)_{i \in [N]}$ sont également distinctes. Les seuls recouvrements de Σ sur lui-même sont donc les recouvrements des règles sur elles-même en ϵ, qui ne sont pas pris en compte dans la définition de la superposition (déf. 5.4). Par conséquent,

$$Crit(\Sigma) = \varnothing$$

L'ensemble $Crit_{\text{pro}}(\Sigma)$ des paires critiques propres est donc vide également. Du corollaire 5.4, on déduit alors le corollaire suivant, aussi donné dans [RV80, cor. 1].

Corollaire 5.11. (Décomposition du pliage/dépliage)
Pour tout système d'équations Σ, on a $\overset{*}{\underset{\Sigma}{\longleftrightarrow}} = \overset{*}{\underset{\Sigma}{\longrightarrow}} \overset{*}{\underset{\Sigma}{\longleftarrow}}$

Autrement dit, toute combinaison de pliages et de dépliages peut se récrire comme une suite de dépliages, suivie d'une suite de pliages. On peut également résumer ce résultat par $\mathbf{ufld} = \mathbf{wuf}$; cette égalité s'entend dans l'interprétation initiale (cf. §5.1.2).

5.4.1 Règles sémantiques linéaires à gauche ou linéaires à droite.

Dans la mesure où les règles de R et les parties gauches des règles de Σ sont construites sur des signatures disjointes, il n'y a pas de superposition propre possible entre R et Σ d'une part, et entre R^{-1} et Σ d'autre part. Les seules superpositions sont impropres et de la forme $\langle x \to \beta \rangle$ sur $\langle \varphi(x_1, \ldots, x_n) \to t \rangle$ en ϵ. Par conséquent,

$$Crit_{\text{pro}}(R \cup R^{-1}, \Sigma) = Crit_{\text{pro}}(\Sigma, R \cup R^{-1}) = \varnothing$$

Du théorème 5.10, on déduit alors le corollaire suivant.

Corollaire 5.12. (Décomposition du dépliage/pliage avec réécriture sémantique)
Pour tout ensemble R de règles linéaires à gauche et tout système d'équations Σ,

$$(\underset{\Sigma}{\longrightarrow} \cup \underset{R}{\longrightarrow} \cup \underset{\Sigma}{\longleftarrow})^* \;=\; \overset{*}{\underset{\Sigma}{\longrightarrow}} \overset{*}{\underset{R}{\longrightarrow}} \overset{*}{\underset{\Sigma}{\longleftarrow}}$$

Cela vaut également si R est linéaire à droite en substituant R^{-1} à R.

Autrement dit, toute combinaison de pliages, de dépliages et de réécritures peut se récrire comme une suite de dépliages, suivie d'une suite de réécritures, et terminée par une suite de pliages. En toute rigueur, nous ne pouvons prétendre $\mathbf{wuf}_R = \mathbf{ufld}_R$ dans le cas où R est linéaire à gauche car la définition de ces transformations emploie $\leftrightarrow_R$ et non $\to_R$. Cependant, dans la mesure où R est constitué d'équations dont la seule contrainte est d'être validées par l'interprétation (cf. §2.5.2, sémantique opérationnelle), cela est peu contraignant en pratique : on peut faire n'importe quelle réécriture compatible avec la sémantique une fois que l'on a opté pour des règles linéaires à gauche ou linéaires à droite. En particulier, ce n'est pas un problème lorsque les règles de R visent à normaliser les termes, car dans ce cas on désire précisément orienter les règles en $\to_R$ au lieu d'employer tout $\leftrightarrow_R$.

Plus formellement, disons qu'un système d'équations $\Sigma' = \{\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) = t'_i \mid i \in [N]\}$ est obtenu respectivement par *dépliage/pliage dirigé* ou *dépliage/pliage faible dirigé* du système d'équations $\Sigma = \{\varphi_i(x_{i,1}, \ldots, x_{i,n_i}) = t_i \mid i \in [N]\}$ ssi :

$$\begin{array}{lll} \Sigma\ \mathbf{dufld}_R\ \Sigma' & \text{ssi} & \text{pour tout } i \in [N] \quad t_i\ (\underset{\Sigma}{\longrightarrow} \cup \underset{R}{\longrightarrow} \cup \underset{\Sigma}{\longleftarrow})^*\ t'_i \\ \Sigma\ \mathbf{dwuf}_R\ \Sigma' & \text{ssi} & \text{pour tout } i \in [N] \quad t_i\ (\underset{\Sigma}{\overset{*}{\longrightarrow}}\ \underset{R}{\overset{*}{\longrightarrow}}\ \underset{\Sigma}{\overset{*}{\longleftarrow}})\ t'_i \end{array}$$

On a alors :

$$\mathbf{dwuf}_R = \mathbf{dufld}_R \quad \text{si} \quad R \text{ est linéaire à gauche ou linéaire à droite}$$

Dans la mesure où $\mathbf{dwuf}_R$ est un cas particulier de $\mathbf{wuf}_R$, on peut dès lors employer les résultats de Kott (cf. §5.2.4) afin de déterminer la correction d'une transformation $\mathbf{dufld}_R$, cas courant des transformations $\mathbf{ufld}_R$. En particulier, nous pouvons traiter à présent (cf. §5.2.4) le cas de la distributivité *ou* de la factorisation (nous ne pouvons employer les deux simultanément car ils ne sont pas linéaires du même coté). Nous pouvons également employer les simplifications par idempotence.

5.4.2 Cas non-linéaire.

Notons que le corollaire 5.12 et par conséquent le théorème 5.10 sont faux si l'hypothèse de linéarite n'est pas satisfaite, même dans le cas de schémas réguliers. Dans l'exemple qui suit, Σ est d'ordre zéro et R n'est linéaire ni à gauche, ni à droite.

- $\Sigma = \{\varphi = h(\varphi)\}$
- $R = \{\langle f(x) \to g(x,x)\rangle, \langle g(x, h(x)) \to x\rangle\}$

Considérons la dérivation suivante :

$$f(\varphi) \underset{R}{\longrightarrow} g(\varphi, \varphi) \underset{\Sigma}{\longrightarrow} g(\varphi, h(\varphi)) \underset{R}{\longrightarrow} \varphi$$

Il est impossible de faire commuter la réduction par Σ pour la repousser vers l'extérieur de la dérivation. En effet, toute dérivation par $\to^*_\Sigma\ \to^*_R\ \leftarrow^*_\Sigma$ est nécessairement de la forme $f(\varphi) \to^n_\Sigma f(h^n(\varphi)) \to_R g(h^n(\varphi), h^n(\varphi)) \leftarrow^{l+m}_\Sigma g(h^{n-l}(\varphi), h^{n-m}(\varphi))$ et il n'est pas possible d'atteindre φ.

La raison de cela est que la partie droite non-linéaire $g(x,x)$ de R et la partie gauche non-linéaire $g(x, h(x))$ ont un unificateur qui est rationnel [Hue76, Cou83] mais qui n'est pas fini : $\{x \mapsto h(h(h(\ldots)))\}$. On ne peut extraire la réduction par Σ, qui reste « piégée » dans ce terme infini[1].

[1] Des résultats partiels nous laissent penser que si aucune dérivation de R ne fait intervenir d'unificateur rationnel infini, on a alors l'identité $(\leftrightarrow_\Sigma \cup \leftrightarrow_R)^* = \to^*_\Sigma\ \leftrightarrow^*_R\ \leftarrow^*_\Sigma$, c'est-à-dire $\mathbf{ufld}_R = \mathbf{wuf}_R$.

Un cas particulier de cette condition couvre les situations courantes ; c'est lorsque toutes les occurrences d'une même variable dans une partie droite ou gauche de règle ont la même longueur, c'est-à-dire lorsque des variables identiques apparaissent à une même profondeur. Le test de cette condition a l'avantage d'être syntaxique et trivial. Il permet par exemple d'employer la règle de distributivité $f(x, g(y,z)) \leftrightarrow g(f(x,y), f(x,z))$ dans les deux sens : la variable x dans le terme non-linéaire $g(f(x,y), f(x,z))$ apparaît à une même profondeur. Il en va de même de l'idempotence $f(x,x) \leftrightarrow x$ (cf. §5.2.4).

5.5 Composer des transformations.

Nous étudions enfin la composition dans un ordre quelconque de plusieurs transformations de pliage/dépliage et d'introduction de définitions auxiliaires. Nous examinons en particulier quand une telle transformation composée est correcte.

Ces définitions auxiliaires sont celles de fonctions intermédiaires, que font nécessairement apparaître les systèmes de transformation de programmes (cf. §1.3.2). Elles correspondent en pratique à des fonctions spécialisées, ou à des configurations, dans le cas de la supercompilation (cf. §1.4.3).

5.5.1 Création de fonctions auxiliaires.

En termes de schémas de programmes, la définition de fonctions auxiliaires consiste à ajouter de nouvelles équations $\langle \varphi(x_1, \dots x_n) = t \rangle$ à un système Σ existant. Ces inconnues additionnelles ne doivent pas être prises en compte dans une comparaison de schémas ni altérer la sémantique : elles ne sont pas *visibles*.

Inconnues auxiliaires.

Pour permettre l'introduction dynamique de ces nouvelles fonctions dans un schéma, nous employons la notion d'*inconnue auxiliaire*. Les définitions suivantes font suite à celles données aux sections §2.5.2 et §5.1.2. Elles sont en majeure partie adaptées de [Cou86, §7].

Sous-système et inconnues auxiliaires. Soit $\Sigma = \{\varphi_i(x_{i,1}, \dots, x_{i,n_i}) = t_i \mid i \in [N]\}$ un système d'équations d'inconnues $\Phi \subset \mathbf{\Phi}$. On définit :

- $\forall i \in [N] \quad \Sigma(\varphi_i) = t_i$
- $\Sigma_{|\Psi} = \{\varphi_i(x_{i,1}, \dots, x_{i,n_i}) = t_i \mid i \in [N], \varphi_i \in \Psi\}$ pour $\Psi \subset \mathbf{\Phi}$
- $\Sigma_{|\Psi}$ est un *sous-système* de Σ ssi $\Sigma_{|\Psi}$ est un système d'équations sur les inconnues Ψ
- Une inconnue $\varphi \in \Phi \cap \Psi$ est appelée *inconnue principale* de Σ. Une inconnue $\varphi \in \Phi \backslash \Psi$ est appelée *inconnue auxiliaire* de Σ.

Comparaison de schémas avec inconnues auxiliaires. Soient $\boldsymbol{C}$ une classe de F-interprétations, Σ et Σ' deux systèmes d'équations sur F avec les inconnues respectives Φ et Φ', et $\Psi \subset \Phi \cap \Phi'$ un ensemble d'inconnues.

- $\Sigma \leqslant_{\boldsymbol{C},\Psi} \Sigma'$ ssi $\forall \boldsymbol{D} \in \boldsymbol{C} \quad \forall \varphi \in \Psi \quad \varphi_{\Sigma,\boldsymbol{D}} \leqslant \varphi_{\Sigma',\boldsymbol{D}}$
- $\Sigma \equiv_{\boldsymbol{C},\Psi} \Sigma'$ ssi $\Sigma \leqslant_{\boldsymbol{C},\Psi} \Sigma'$ et $\Sigma \geqslant_{\boldsymbol{C},\Psi} \Sigma'$

Si $\Sigma_{|\Psi}$ et $\Sigma'_{|\Psi}$ sont des *sous-systèmes* respectifs de Σ et Σ', alors

- $\Sigma \leqslant_{\boldsymbol{C},\Psi} \Sigma'$ ssi $\Sigma_{|\Psi} \leqslant_{\boldsymbol{C}} \Sigma'_{|\Psi}$
- $\Sigma \equiv_{\boldsymbol{C},\Psi} \Sigma'$ ssi $\Sigma_{|\Psi} \equiv_{\boldsymbol{C}} \Sigma'_{|\Psi}$

On omet l'indice $\boldsymbol{C}$ lorsque c'est l'ensemble de toutes les F-interprétations continues, ou bien lorsque $\boldsymbol{C} = \boldsymbol{M}_{\Omega}^{\infty}(F)$.

Transformation et correction de schémas avec inconnues auxiliaires. Soient $\boldsymbol{C}$ une classe de F-interprétations et Ψ un ensemble d'inconnues.

- Une transformation de Σ par $\boldsymbol{trans}$ est $\boldsymbol{C}$,Ψ-*correcte* ssi $\Psi \subset \Phi$ et pour tout Σ' d'inconnues Φ' tel que $\Psi \subset \Phi'$ et $\Sigma\ \boldsymbol{trans}\ \Sigma'$, alors $\Sigma \equiv_{\boldsymbol{C},\Psi} \Sigma'$.
- Une transformation $\boldsymbol{trans}$ est $\boldsymbol{C}$,Ψ-*correcte* ssi pour tous systèmes d'équations Σ et Σ' d'inconnues respectives Φ et Φ' telles que $\Psi \subset \Phi \cap \Phi'$, si $\Sigma\ \boldsymbol{trans}\ \Sigma'$, alors $\Sigma \equiv_{\boldsymbol{C},\Psi} \Sigma'$.
- Si une transformation de Σ par $\boldsymbol{trans}$ est correcte, alors elle est $Unk(\Sigma)$-correcte.

Une transformation Ψ-*correcte* est une transformation $\boldsymbol{M}_\Omega^\infty(F)$,$\Psi$-*correcte*.

Introduction de nouvelles définitions.

Maintenant posées ces notations, nous pouvons définir la transformation qui exprime l'introduction de nouvelles définitions.

Définition 5.7. (Introduction d'une définition auxiliaire)
Soient Φ un ensemble fini d'inconnues, $\varphi \notin \Phi$, et $t, \varphi(x_1, \ldots, x_n) \in M(F \cup \Phi \cup \{\varphi\}, \{x_1, \ldots, x_n\})_{\sigma(\varphi)}$. L'*introduction de la définition auxiliaire* $\langle \varphi(x_1, \ldots, x_n) = t \rangle$ est la transformation sur les schémas de programme qui est définie par :

- $\Sigma\ \mathbf{ndef}_{\langle \varphi(x_1,\ldots,x_n)=t \rangle}\ \Sigma'$ ssi $Unk(\Sigma) \subset \Phi$ et $\Sigma' = \Sigma \cup \{\varphi(x_1, \ldots, x_n) = t\}$
- $\Sigma\ \mathbf{ndef}\ \Sigma'$ ssi il existe $\varphi(x_1, \ldots, x_n)$ et t tels que $\Sigma\ \mathbf{ndef}_{\langle \varphi(x_1,\ldots,x_n)=t \rangle}\ \Sigma'$.
- $\Sigma\ \mathbf{ndefs}\ \Sigma'$ ssi $\Sigma\ \mathbf{ndef}^*\ \Sigma'$

La transformation $\mathbf{ndef}_{\langle \varphi(x_1,\ldots,x_n)=t \rangle}$ est déterministe : un seul Σ' vérifie $\Sigma\ \mathbf{ndef}_{\langle \varphi(x_1,\ldots,x_n)=t \rangle}\ \Sigma'$.

Proposition 5.13. (Correction de l'introduction de nouvelles définitions)
Pour tout système d'équation Σ d'inconnues Φ, toute transformation $\Sigma\ \mathbf{ndefs}\ \Sigma'$ est Φ-correcte.

Démonstration. Pour tout $\langle \varphi(x_1, \ldots, x_n) = t \rangle \in \Sigma$, le terme t appartient à $M(F \cup \Phi, X)$. Or $\Sigma_{|\Phi} = \Sigma'_{|\Phi}$, donc $\varphi_\Sigma = \varphi_{\Sigma'}$ dans la F-algébre libre continue $M_\Omega^\infty(F, X)$. Par conséquent, $\Sigma \equiv_\Phi \Sigma'$. $\square$

En pratique, tous les langages ne permettent pas l'emploi de fonctions auxilaires, c'est-à-dire en fait de fonctions dont la *visibilité* (scope) peut être restreinte. En ML par exemple, ce type de fonctions est introduit par les constructions « `let` *decs* `in` *exp* `end` » et « `local` *decs* `in` *decs'* `end` », qui limitent la portée des déclarations *decs*. Il n'existe pas en revanche de tels mécanismes dans des langages comme LISP, SCHEME, PROLOG, C. . . , et l'on doit se résoudre en pratique à générer un nouveau nom, suffisamment « étrange » pour qu'il ne risque pas d'entrer en collision avec d'autres fonctions écrites par le programmeur, ou par un autre système de transformation. En toute rigueur, pour ces langages, aucune transformation qui crée de nouvelles fonctions n'est correcte. Néanmoins, certaines extensions remédient à cette carence [MS91].

5.5.2 Dérivation de schémas.

En termes de schémas de programme, composer des transformations forme une suite $\Sigma_0, \Sigma_1, .., \Sigma_n$ de systèmes d'équations. Avant de l'étudier , nous posons quelques notations concernant les mécanismes de passage d'un schéma Σ_i à un schéma Σ_{i+1}.

Il nous faut auparavant poser quelques notations afin d'exprimer précisément les mécanismes de passage d'un schéma Σ_i à un schéma Σ_{i+1}.

Définition 5.8. (Dérivation de schémas)
Soient S un ensemble de règles sur $M(F \cup \Phi, X)$. Pour tout $\nabla \in \mathit{Deriv}(S)$, on définit les relations suivantes sur les systèmes d'équations d'inconnues Φ.

- $\Sigma \rightarrow_{\varphi,\nabla} \Sigma'$ ssi $\Sigma(\varphi) \rightarrow_{\nabla}^{*} \Sigma'(\varphi)$ et $\Sigma_{|\Phi\setminus\{\varphi\}} = \Sigma'_{|\Phi\setminus\{\varphi\}}$
- $\Sigma \rightarrow_{\varphi,S} \Sigma'$ ssi $\exists \nabla \in \mathit{Deriv}(S) \quad \Sigma \rightarrow_{\varphi,\nabla} \Sigma'$
- $\Sigma \rightarrow_{S} \Sigma'$ ssi $\exists \varphi \in \Phi \quad \Sigma \rightarrow_{\varphi,S} \Sigma'$

Cette définition s'emploie en particulier pour le dépliage $S = \Sigma$, le pliage $S = \Sigma^{-1}$, ou la réécriture selon des lois algébriques $S = R$.

5.5.3 Séquence de transformations.

Nous définissons une *séquence de transformations* comme une suite de transformations élémentaires constituées de pliages, de dépliages, de réécritures selon les lois algébriques, ou de définitions de fonctions auxiliaires. À partir d'un schéma de départ, une séquence de transformation construit ainsi une suite $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$ de systèmes d'équations.

Par ailleurs, pour passer d'une étape Σ_i à une étape Σ_{i+1}, nous nous autorisons, en cas de pliage ou de dépliage d'une inconnue, à employer non seulement les définitions de Σ_i, mais toutes celles déjà calculées : $\Sigma_0, \ldots, \Sigma_i$. Cette transformation diffère a priori d'un simple enchaînement de transformations de pliage/dépliage qui, en quelque sorte, « n'a pas de mémoire » : pour passer de Σ_i à Σ_{i+1}, les pliages et dépliages sont en ce cas réalisés avec le seul schéma Σ_i.

D'autre part, on peut noter que les transformations ne sont généralement ni confluentes[2], ni réversibles[3]. Par conséquent, un choix de transformation dans un algorithme est le plus souvent définitif, et il vaut mieux disposer non seulement du maximum de choix possibles, mais aussi de transformations de « faible granularité » (les systèmes d'indices plus faibles ont subi a priori moins de dépliages et sont donc plus réduits) qui permettent des avancées modérées mais contrôlées.

Une séquence de transformations laisse ainsi une plus grande latitude dans les stratégies d'un système automatique de transformation de programmes.

Définition 5.9. (Séquence de transformations)
Un système d'équations Σ' est obtenu par *séquence de transformations* de Σ ssi il existe une suite de schémas $(\Sigma_i)_{i\in[0..n]}$ telle que $\Sigma = \Sigma_0$, $\Sigma_n = \Sigma'$, et :

[2] Plus pour des raisons pratiques que théoriques.
[3] Une fois le schéma $\Sigma = \langle\varphi = f(\varphi)\rangle$ déplié en $\Sigma' = \langle\varphi = f(f(\varphi))\rangle$, il n'est plus possible de retrouver Σ à partir de Σ' par simples pliages ou dépliages.

- $\forall i \in [0..n-1] \quad \begin{cases} S_i = \bigcup_{j\in[i]} \Sigma_i \\ \Sigma_i \ (\rightarrow_{S_i} \cup \leftrightarrow_R \cup \leftarrow_{S_i}) \ \Sigma_{i+1} \ \text{ ou } \ \Sigma_i \ \mathbf{ndef} \ \Sigma_{i+1} \end{cases}$

Une séquence de transformations de Σ à Σ' est notée $\Sigma \ \mathbf{sequfld}_R \ \Sigma'$. Elle est correcte ssi $\Sigma \equiv_R \Sigma'$.

Il faut noter qu'une transformation de pliage/dépliage $\mathbf{ufld}_R$ est un cas particulier de séquence de transformations, où pliages et dépliages s'effectuent par rapport au seul schéma de départ. Par conséquent, les résultats de correction de $\mathbf{ufld}_R$ ne s'appliquent pas directement à une séquence de transformations.

Néanmoins, la proposition suivante exprime que toute séquence de transformations peut se mettre sous une forme normale, comme la composition de définitions auxiliaires suivies d'une transformation de pliage/dépliage $\mathbf{ufld}_R$. La proposition 5.13 assure alors la correction des définitions auxiliaires et permet donc de ramener le problème à celui de la correction d'une transformation $\mathbf{ufld}_R$.

Proposition 5.14. (Décomposition d'une séquence de transformation)
Il existe une séquence de transformations de Σ en Σ' ssi $\Sigma \ \mathbf{ndefs} \ \mathbf{ufld}_R \ \Sigma'$.

Démonstration. Nous montrons d'abord que, dans une séquence de transformations $(\Sigma_i)_{i\in[0..n]}$, il est possible de faire permuter de la droite vers la gauche toutes les introductions de nouvelles définitions, afin de les repousser en tête de séquence :

- Si $\Sigma_i \rightarrow_{\varphi,u,r,\sigma} \Sigma_{i+1} \ \mathbf{ndef}_{\langle \varphi'(x'_1,\dots,x'_{n'})=t'\rangle} \ \Sigma_{i+2}$ avec $\langle \varphi(x_1,\dots x_n) = t\rangle \in \Sigma_i$ et $t \rightarrow_{u,r,\sigma} s$ alors $\Sigma_{i+2} = \Sigma_{i|Unk(\Sigma_i)\setminus\{\varphi\}} \cup \{\varphi(x_1,\dots x_n) = s, \varphi'(x'_1,\dots,x'_{n'}) = t'\}$, donc $\Sigma_i \ \mathbf{ndef}_{\langle \varphi'(x'_1,\dots,x'_{n'})=t'\rangle} \ \Sigma'_{i+1} \rightarrow_{\varphi,u,r,\sigma} \Sigma_{i+2}$ avec $\Sigma'_{i+1} = \Sigma_i \cup \{\varphi(x_1,\dots,x_n) = t\}$. En effet $\varphi \notin \Sigma_i$ et $t, \varphi(x_1,\dots,x_n) \in M(F \cup Unk(\Sigma_i), X)$ car $\varphi \notin \Sigma_{i+1}$ et $Unk(\Sigma_i) = Unk(\Sigma_{i+1})$. Donc, par récurrence sur la longueur de la séquence de transformation, toutes les transformations $\mathbf{ndef}$ peuvent être remontées en tête de séquence.

Il ne reste donc à considérer que le cas des transformations $\Sigma_i \ (\rightarrow_{S_i} \cup \leftrightarrow_R \cup \leftarrow_{S_i}) \ \Sigma_{i+1}$. Nous montrons par récurrence sur $n \in \mathbb{N}$ que pour tout $(\Sigma_i)_{i\in[0..n]}$, pour tout $i \in [n]$, il existe une dérivation $\Sigma_0 \rightarrow^*_{\Sigma_0 \cup R \cup R^{-1} \cup \Sigma_0^{-1}} \Sigma_i$. Notons $\Phi = Unk(\Sigma_0)$ l'ensemble des inconnues du schéma Σ_0, et $S'_0 = \Sigma_0 \cup R \cup R^{-1} \cup \Sigma_0^{-1}$. On a donc $\rightarrow_{S'_0} = \rightarrow_{\Sigma_0} \cup \leftrightarrow_R \cup \leftarrow_{\Sigma_0}$ et $(S'_0)^{-1} = S'_0$.

Le cas est trivial pour $n = 0$. Lorsque $n \geqslant 1$, pour tout $i \in [n-1]$ on a $\Sigma_0 \rightarrow^*_{S'_0} \Sigma_i$ par l'hypothèse de récurrence. En particulier, $\Sigma_0 \rightarrow^*_{S'_0} \Sigma_{n-1}$. D'autre part, $\Sigma_{n-1} \ (\rightarrow_{\Sigma_i} \cup \leftrightarrow_R \cup \leftarrow_{\Sigma_i}) \ \Sigma_n$ pour un certain $i \in [0..n-1]$. Il faut envisager trois cas :

1. Supposons $\Sigma_{n-1} \rightarrow_{\varphi,u,r,\sigma} \Sigma_n$ avec $r = \langle \psi(x_1,\dots,x_m) = t\rangle \in \Sigma_i$. On a donc $\Sigma_{n-1}(\varphi)/u = \sigma.\psi(x_1,\dots,x_m)$ et $\Sigma_n(\varphi)/u = \sigma.t$. Soit $r_0 = \langle \psi(x_1,\dots,x_m) \rightarrow \Sigma_0(\psi)\rangle \in \Sigma_0$. En employant une nouvelle fois l'hypothèse de récurrence, on sait qu'il existe $\Sigma_0 \rightarrow_{(\varphi,\nabla_\varphi)_{\varphi\in\Phi}} \Sigma_i$ avec $\forall \varphi \in \Phi$ $\nabla_\varphi \in Deriv(S'_0)$. Or $\sigma.\psi(x_1,\dots,x_m) \rightarrow_{\epsilon,r_0,\sigma} \sigma.\Sigma_0(\psi) \rightarrow_{\sigma.\nabla_\psi} \sigma.\Sigma_i(\psi) = \sigma.t$. Posons $\nabla' = (\epsilon, r_0, \sigma); \sigma.\nabla_\psi \in Deriv(S'_0)$, alors $\Sigma_{n-1}(\varphi)/u \rightarrow_{\nabla'} \Sigma_n(\varphi)/u$. Par conséquent $\Sigma_0 \rightarrow^*_{S'_0} \Sigma_{n-1} \rightarrow_{\varphi,\nabla''} \Sigma_n$ avec $\nabla'' = u \cdot \nabla' \in Deriv(S'_0)$.
2. Si $\Sigma_{n-1} \rightarrow_{\varphi,u,r,\sigma} \Sigma_n$ avec $r \in R \cup R^{-1}$, alors $\Sigma_{n-1} \leftrightarrow_R \Sigma_n$, donc $\Sigma_0 \rightarrow^*_{S'_0} \Sigma_{n-1} \rightarrow_{S'_0} \Sigma_n$.
3. Supposons $\Sigma_{n-1} \leftarrow_{\varphi,u,r,\sigma} \Sigma_n$ avec $r = \langle \psi(x_1,\dots,x_m) = t\rangle \in \Sigma_i$. Si l'on permute Σ_{n-1} et Σ_n dans le point (1.) ci-dessus, on obtient une dérivation $\Sigma_{n-1}(\varphi)/u \leftarrow_{\nabla'} \Sigma_n(\varphi)/u$ avec $\nabla' \in Deriv(S'_0)$, donc $\Sigma_{n-1}(\varphi)/u \rightarrow_{(\nabla')^{-1}} \Sigma_n(\varphi)/u$ avec $(\nabla')^{-1} \in Deriv(S_0'^{-1}) = Deriv(S'_0)$. Par conséquent $\Sigma_0 \rightarrow^*_{S'_0} \Sigma_{n-1} \rightarrow_{\varphi,\nabla''} \Sigma_n$ avec $\nabla'' = u \cdot (\nabla')^{-1} \in Deriv(S'_0)$.

Pour tout $n \in \mathbb{N}$, on a donc $\Sigma_0 \rightarrow_{\Sigma_0 \cup R \cup R^{-1} \cup \Sigma_0^{-1}} \Sigma_n$, c'est-à-dire $\Sigma_0 \ \mathbf{ufld}_R \ \Sigma_n$. La réciproque est triviale puisque $\mathbf{ufld}_R$ est un cas particulier de séquence de transformations. □

Il faut noter toutefois que la conversion d'une séquence de transformations $(\Sigma_i)_{i\in[0..n]}$ en une transformation de pliage/dépliage Σ_0 $\mathbf{ufld}_R$ Σ_n décrite dans la démonstration de la proposition 5.14 n'est pas « optimale ». En particulier, si l'on est certain à partir d'une étape i que l'on n'employera jamais plus les règles des systèmes d'équation $(\Sigma_j)_{j<i}$, il vaut mieux montrer la correction de $(\Sigma_j)_{j\in[0..i]}$, puis prendre Σ_i comme nouveau schéma de départ, et montrer la correction de $(\Sigma_j)_{j\in[i..n]}$. On évite ainsi des pliages et des dépliages supplémentaires pour ramener le problème en termes de dérivations de Σ_0. Plus généralement, il serait intéressant d'étudier formellement comment tester la correction de manière *incrémentale*, sans avoir à faire explicitement la remontée des transformations $\mathbf{ndef}$ et la conversion en dérivations de Σ_0.

Notons enfin que les dérivations de schémas sont identiques pour les cas algébriques et réguliers. Il n'y a pas ici de différence fondamentale entre les deux formulations (cf. §2.5.7).

Conclusion du chapitre.

Nous avons présenté une variante sur le thème de la commutativité des systèmes de réécriture linéaires à gauche (prop. 5.3). Nous avons pour cela employé une notion de dérivation abstraite (déf. 5.1, 5.2, 5.3) et de paire critique propre (déf. 5.5).

Avec les mêmes techniques, nous avons ensuite démontré un lemme voisin sur les systèmes confluents (lemme 5.6), qui permet d'établir un théorème de forme normale sur les dérivations entrelacées de deux systèmes de réécriture (th. 5.10).

Une application de ce théorème montre qu'il suffit d'une linéarité à gauche ou d'une linéarité à droite pour ramener une transformation quelconque à une transformation de dépliage/pliage faible (§5.4). Cette extension permet en particulier de traiter le cas des simplifications qui emploient la distributivité, la factorisation, et l'idempotence, simplifications qui sont impossibles avec la condition ordinaire de linéarité à gauche et à droite. Par ailleurs, nous avons donné un exemple de transformation qui emploie une règle qui n'est linéaire ni à gauche, ni à droite, et qui ne peut pas se ramener à une transformation de dépliage/pliage faible (cf. §5.4.2). Déterminer les conditions pour lesquelles on peut lever l'hypothèse de linéarité reste un problème ouvert.

Nous avons défini une notion de séquence de transformations, qui permet d'utiliser, dans des pliages ou des dépliages, les schémas intermédiaires obtenus à toutes les étapes précédentes (déf. 5.9). Nous avons montré comment ramener une telle séquence sous une forme normale (prop. 5.14), dont la correction peut être traitée à l'aide des résultats précédents. Nous avons employé pour cela une notion abstraite de dérivation de schéma (déf. 5.8).

Au chapitre suivant, nous étudions une classe d'algorithmes qui peut permettre de piloter des transformations de pliage et de dépliage.

Chapitre 6

Algorithme et Terminaison

Aucune stratégie ne guide l'emploi des transformations définies au chapitre 5 ; nous pouvons simplement déterminer, lorsque l'on nous donne explicitement une transformation de pliage/dépliage, si elle est correcte ou non. Le rôle d'un système *automatique* de transformation de programme est d'établir une stratégie d'application des transformations qui permette, étant donné un programme, de produire en un temps fini un programme transformé.

Nous ne proposons pas dans ce chapitre une stratégie générale qui puisse s'appliquer à tout langage de programmation, car c'est un problème trop spécifique. En revanche, nous présentons différents composants formels qui constituent un squelette pour un système automatique de transformation de programme. Cette ossature doit ensuite être garnie des nécessités propres à chaque langage.

Ces composants répondent à deux types de problème : pouvoir enchaîner des transformations élémentaires en garantissant la terminaison du processus global de transformation, et permettre d'effectuer le maximum de transformations, tant que cela reste compatible avec la terminaison.

Nous abordons le premier point par l'étude de l'*algorithme de redémarrage généralisé*, employé notamment en supercompilation. Cet algorithme est contrôlé par un *critère d'arrêt* quelconque ; dans un deuxième temps, nous examinons comment le choisir.

Organisation du chapitre.

§6.1 Nous étudions dans cette première section l'algorithme de redémarrage généralisé, qui permet de piloter une séquence de transformations. Nous démontrons sa terminaison, parfois négligée, et examinons la nature de la *fonction de généralisation*.

§6.2 D'autre part, nous analysons la nature et les qualités des critères d'arrêt, qui peuvent équiper un algorithme d'évaluation partielle. En particulier, nous examinons les mérites d'un critère d'arrêt basé sur le théorème de Kruskal.

Ainsi s'achève le corps de ce document. Il ne restera plus à la conclusion qu'à retracer le chemin parcouru, proposer quelques perspectives, et commenter le « bien-fondé » de l'évaluation partielle.

6.1 Redémarrage généralisé.

Les constructions de la section §5.5 ne permettent pas de réaliser un outil automatique qui fait des transformations de programme. On suppose une séquence de transformations *donnée* et l'on statue sur sa correction.

Décider quelle succession de transformations élémentaires sont appliquées au programme initial, et décréter quand la transformation est terminée est le rôle d'un *algorithme de transformation*. Dans cette section, nous étudions un mécanisme de ce type, dont l'objectif est de permettre le maximum de transformations tout en respectant un critère d'arrêt (cf. §6.2). Il s'agit de l'algorithme de *redémarrage généralisé*.

Il ne s'agit en fait pas d'un algorithme proprement dit mais d'un outil de construction d'algorithme. Il ne décrit pas explicitement comment piloter la construction d'une séquence de transformations ; pour le mettre en œuvre pratiquement, il faut notamment spécifier une fonction qui réalise des enchaînements de transformations, ainsi qu'un critère de terminaison.

L'algorithme de redémarrage généralisé a principalement été employé en supercompilation (cf. §1.4.3, algorithme) et en spécialisation (cf. §1.3.4, terminaison), avec toutefois certaines libertés quant à la preuve de sa terminaison [Tur88, Has88, Sah91a, WCRS91]. Nous revenons sur ce point et étudions également la nature de la fonction de généralisation.

6.1.1 Aperçu.

Pour donner un aperçu de l'algorithme de redémarrage généralisé, nous reprenons l'exemple de la fonction factorielle, à l'aide cette fois-ci d'une définition qui fait intervenir un accumulateur.

$$\begin{array}{lcl} \mathbf{fac}(n) & = & \mathbf{facc}(n, 1, 1) \\ \mathbf{facc}(n, i, r) & = & \mathsf{if}(\mathsf{ge}(i, n), r, \mathbf{facc}(n, \mathsf{add1}(i), \mathsf{mul}(\mathsf{add1}(i), r))) \end{array}$$

Dans cette formulation, i varie entre 1 et n, et r prend les valeurs successives de $i!$ au cours de la dérivation de $\mathbf{fac}(n)$. Dans le contexte d'une évaluation partielle, le terme $\mathbf{facc}(n, 1, 1)$ apparaît comme un appel fonctionel de **facc** à *spécialiser* (cf. §1.3) sur les valeurs statiques $(1, 1)$ des deux derniers arguments. On effectue pour cela quelques dépliages et réécritures selon les lois algébriques.

$$\begin{array}{rcl} \mathbf{fac}(n) & \rightarrow & \underline{\mathbf{facc}(n, 1, 1)} \\ & \rightarrow & \mathsf{if}(\mathsf{ge}(1, n), 1, \mathbf{facc}(n, \underline{\mathsf{add1}(1)}, \underline{\mathsf{mul}(\mathsf{add1}(1), 1)})) \\ & \xrightarrow{*} & \mathsf{if}(\mathsf{ge}(1, n), 1, \underline{\mathbf{facc}(n, 2, 2)}) \end{array}$$

Pour poursuivre cette dérivation, il faudrait à présent déplier $\mathbf{facc}(n, 2, 2)$. Cependant, les arguments de **facc** semblent croître et l'on n'est pas certain que ces dérivations puissent se terminer. Faisons l'hypothèse qu'un *critère d'arrêt* (cf. §6.2) déclare qu'il ne faut pas poursuivre plus avant.

Remarquons tout de même que les deuxième et troisième arguments dans $\mathbf{facc}(n, 1, 1) \rightarrow^* \mathbf{facc}(n, 2, 2)$ sont identiques. S'il n'est pas possible de spécialiser par rapport à $(1, 1)$, peut-être peut-on malgré tout faire une supercompilation (on ne peut plus alors tout à fait parler de spécialisation) à partir de la configuration $\mathbf{facc}(n, k, k)$. On espère ainsi éliminer l'un des arguments de **facc** pour obtenir une définition récursive qui ne fait plus intervenir que deux arguments au lieu de trois. Il faut pour cela introduire une fonction auxiliaire (cf.

§5.5.1). On a alors les définitions suivantes.

$$\begin{aligned}
\mathbf{fac}(n) &= \mathbf{facc}'(n, 1)\\
\mathbf{facc}'(n, k) &= \mathbf{facc}(n, k, k)\\
\mathbf{facc}(n, i, r) &= \mathsf{if}(\mathsf{ge}(i, n), r, \mathbf{facc}(n, \mathsf{add1}(i), \mathsf{mul}(\mathsf{add1}(i), r)))
\end{aligned}$$

Faisons subir à $\mathbf{facc}'$ quelques réductions.

$$\begin{aligned}
\mathbf{facc}'(n, k) &\rightarrow \underline{\mathbf{facc}(n, k, k)}\\
&\rightarrow \mathsf{if}(\mathsf{ge}(k, n), r, \underline{\mathbf{facc}(n, \mathsf{add1}(k), \mathsf{mul}(\mathsf{add1}(k), k))})
\end{aligned}$$

Au vu à nouveau des deuxième et troisième arguments (le premier, n, reste constant), on peut imaginer qu'un critère d'arrêt signale encore un risque de croissance, structurelle cette fois-ci : $k \rightarrow \mathsf{add1}(k)$ et $k \rightarrow \mathsf{mul}(\mathsf{add1}(k), k)$. Par conséquent, on ne doit pas faire de dépliage plus avant.

Cependant, un pliage de $\mathbf{facc}(n, \mathsf{add1}(k), \mathsf{mul}(\mathsf{add1}(k), k))$ sur $\mathbf{facc}'$ est également impossible : tous les arguments de **facc** finissent par être différents au cours d'une dérivation. La conclusion sur cet exemple est qu'il n'était en fait pas possible de simplifier davantage la définition initiale de **fac**.

Le *redémarrage* est l'opération qui consiste, au cours d'une suite de transformation, à renoncer à transformer un terme au bout de quelques dérivations supplémentaires pour reprendre complètement les calculs à partir de ce point, sur une autre configuration. Ici $\mathbf{facc}(n, 1, 1)$ est développé jusqu'à ce que l'on rencontre $\mathbf{facc}(n, 2, 2)$, puis oublié afin de redémarrer avec le terme $\mathbf{facc}'(n, k) = \mathbf{facc}(n, k, k)$.

Le redémarrage est *généralisé* car le terme $\mathbf{facc}(n, k, k)$ est *plus général* que $\mathbf{facc}(n, 1, 1)$: il existe une substitution $\sigma = \{k \mapsto 1\}$ telle que $\sigma.\mathbf{facc}(n, k, k) = \mathbf{facc}(n, 1, 1)$ (cf. §1.4.3, transformations). On note cela $\mathbf{facc}(n, k, k) \leqslant \mathbf{facc}(n, 1, 1)$. Notons que dans cet exemple $\mathbf{facc}(n, k, k)$ est également plus général que $\mathbf{facc}(n, 2, 2)$: c'est un dénominateur commun entre $\mathbf{facc}(n, 1, 1)$ et $\mathbf{facc}(n, 2, 2)$.

Dans le cas d'un schéma régulier, la généralisation s'exprime à l'aide des variables syntaxiques du langage et non plus des variables algébriques de schéma. On a par exemple $\mathsf{call}_3(\mathbf{facc}, \mathsf{var}(n), 1, 1) \geqslant \mathsf{call}_3(\mathbf{facc}, \mathsf{var}(n), \mathsf{var}(k), \mathsf{var}(k)) \geqslant \mathsf{call}_3(\mathbf{facc}, \mathsf{var}(n), \mathsf{var}(i), \mathsf{var}(r))$.

6.1.2 L'algorithme.

Après cet aperçu, nous donnons une définition précise de l'algorithme de redémarrage généralisé. En pratique, il n'est jamais présenté ainsi, mais habillé de spécificités propres au langage de programmation considéré.

Il faut tout d'abord préciser un *bon préordre* $\trianglelefteq$, qui permet d'interrompre une suite infinie $(q_n)_{n \in \mathbb{N}}$ en détectant par une énumération finie deux indices $i < n \in \mathbb{N}$ tels que $q_i \trianglelefteq q_n$. C'est un test pessimiste qui détermine lorsqu'il y a un *risque de non-terminaison* : il ne permet pas d'entrer dans une boucle infinie, mais certains calculs, pourtant finis, peuvent être interrompus prématurément parce qu'on estime qu'il y a une possibilité pour qu'ils soient infinis.

Puis, lors d'une séquence de transformations de programme, si un risque de boucle entre l'étape courante q_n et une étape antérieure q_i est détecté par le test $\trianglelefteq$, on reprend totalement le calcul au niveau de q_i en *généralisant* la configuration à transformer (cf. §1.4.3, transformations). Les transformations se poursuivent ensuite sur la configuration plus générale.

Revenir ainsi à une étape antérieure est analogue en quelque sorte au *retour arrière* (backtrack) employé dans le parcours d'un SLD-arbre en PROLOG. La seule différence est que ce retour arrière-ci ne remonte pas nécessairement à son père, c'est-à-dire q_{n-1}, mais peut remonter à un ancêtre q_i quelconque.

En pratique, la nouvelle valeur de départ en i est obtenue par $q'_i = \gamma(q_i, q_n)$ où γ est une fonction à préciser dans l'algorithme, qui doit qui calculer une configuration plus générale que q_i (cf. §6.1.3). Un préordre bien fondé $\leq$ relie ces configurations de généralité croissante afin d'assurer la finitude du nombre d'expressions considérées. Dans l'exemple de la section §6.1.1, ce préordre $\leq$ est la relation $\leqslant$ de plus grande généralité sur les termes (cf. §N.16).

L'arrêt survient lorsque la généralisation est stable, c'est-à-dire lorsque q'_i et q_i sont équivalents selon $\leq$. Le terme le plus général sur lequel puisse se produire l'arrêt est une variable x.

En voici une définition plus formelle et plus abstraite, qui ne conserve que les informations qui intéressent la terminaison.

Définition 6.1. (Algorithme de redémarrage généralisé)
Soient E un ensemble, $\trianglelefteq$ un bon préordre sur E, $\leq$ un préordre bien fondé sur E, $\varrho : E \to E$ une application, et $\gamma : E \times E \to E$ une application $\leq$-décroissante en son premier argument. Pour tout $q \in E$, on calcule l'élément $\varrho_\gamma(q)$ de E par l'algorithme suivant :

1. $i := 1$; $q_{\text{aux}} := q$
2. répéter
3. $\quad n := i$; $q_n := q_{\text{aux}}$
4. $\quad$ tant que $\nexists\, i \in [1..n]\ \ q_i \trianglelefteq q_n$ répéter $q_{n+1} := \varrho(q_n)$; $n := n + 1$
5. $\quad q_{\text{aux}} := \gamma(q_i, q_n)$
6. jusqu'à ce que $q_{\text{aux}} \cong q_i$
7. $\varrho_\gamma(q) := q_n$

La notation $x := y$ désigne l'affectation de la valeur y à la variable x, et $x \cong y$ l'équivalence $x \leq y$ et $x \geq y$.

Le nombre n est l'indice de l'état courant, et i est l'indice de l'étape où a lieu un *redémarrage*. Les états $(q_j)_{j \leqslant i}$ sont préservés, et l'on reprend à partir de q_i le processus de transformation représenté par ϱ. Il faut noter que l'algorithme de redémarrage généralisé n'est pas lié au formalisme des schémas de programmes, ni même à un formalisme algébrique.

Le terme *généralisé* provient des mises en œuvre pratiques de cet algorithme, dans lesquelles $\leq$ est souvent la relation $\leqslant$ de plus grande généralité sur des termes algébriques. Cependant, toute autre interprétation de $\leq$ et γ reste possible. En particulier, on peut également faire intervenir le typage [WCRS91].

Dans [Tur88, Has88, Sah91a, WCRS91], la démonstration que ce processus se termine se ramène à ceci : « le nombre de termes $\leq$-inférieurs à q_i est fini, car $\leq$ est une relation bien fondée, donc l'algorithme se termine ». Ce raisonnement semble faire l'*hypothèse implicite* qu'une fois qu'un redémarrage à eu lieu à une étape i, tous les redémarrages suivants le sont aussi relativement à cette même étape.

Tel que nous avons posé le problème, ce n'est pas nécessairement le cas ; nous pourrions avoir une suite infinie de redémarrage successifs à des étapes d'indice croissant. Rien n'interdit non plus que le redémarrage suivant ait lieu à une étape d'indice inférieur à i.

L'*explication vraisemblable* de ce raccourci tient à ce que les états $q'_i, q'_{i+1}, \dots$ parcourus après un redémarrage en i sont *implicitement supposés* être plus généraux que leurs antécédants $q_i, q_{i+1}, \dots, q_n$. En d'autres termes, les transformations sont supposées monotones par rapport à la généralisation. Sous cette hypothèse, si un nouveau redémarrage devait se produire à un rang $j > i$, il aurait en fait déjà eu lieu au moment de l'énumération $q_i, \dots, q_j, \dots q_n$.

Néanmoins, même cette justification incomplète n'apparaît pas dans les références que nous avons citées. A fortiori, une démonstration de la monotonie des transformations par rapport à la généralisation n'apparaît pas non plus.

Nous ne formalisation pas davantage ces notions car, heureusement et malgré tout, cet algorithme se termine toujours, sans hypothèses supplémentaires. C'est l'objet de la proposition suivante.

Proposition 6.1. (Terminaison de l'algorithme de redémarrage généralisé)
L'algorithme de la définition 6.1 se termine pour tout $q \in E$.

Démonstration. La terminaison de cet algorithme repose sur le lemme de König, qui dit que tout arbre infini tel que chaque nœud n'a qu'un nombre fini de fils contient un chemin infini[1].

Nous associons à l'exécution de cet algorithme un arbre étiqueté à chaque nœud par un état de E, à l'exception du nœud racine, qui ne porte pas d'étiquette. C'est l'analogue de l'arbre définit par le parcours incomplet de l'arbre de SLD-résolution en PROLOG.

On déplace une marque (une occurrence) sur les nœuds de l'arbre, qui détermine un « nœud courant ». Le point de départ de la construction est un arbre constitué d'un seul nœud racine ; c'est également le nœud courant initial.

Pour chaque affectation $q_n := \dots$ un nouveau nœud portant la nouvelle valeur de q_n est ajouté comme fils du nœud courant (à droite des frères éventuels), et devient ensuite le nouveau nœud courant. L'itération de la boucle intérieure (4.) construit ainsi une branche portant les valeurs $q_i, \dots, q_n$.

Lorsqu'on arrive à la fin d'une itération de la boucle externe (2.–6.), c'est-à-dire lorsqu'on a trouvé un indice i tel que $q_i \trianglelefteq q_n$, le nœud courant devient le père du nœud marqué par cet q_i.

Soit l'algorithme s'arrête là parce que $q_i \cong \gamma(q_i, q_n)$; soit il effectue une nouvelle fois les instructions (2.–6.), ce qui a pour effet de faire germer une branche sœur de la branche $q_i, \dots, q_n$.

Il faut noter deux points importants de cette construction :

(1) Pour tout nœud différent de la racine et marqué par un état q_i, s'il a un nœud frère à droite, alors ce frère est étiqueté par un état q'_i calculé par $q'_i := \gamma(q_i, q_n)$. De plus, cet état vérifie $q'_i \leq q_i$.
Il n'est pas possible que $q'_i \cong q_i$, car alors l'algorithme se serait arrêté auparavant. Par conséquent, on a $q'_i < q_i$ et l'on sait que l'exécution de l'algorithme s'est poursuivie. La suite des frères est donc strictement décroissante.
Par conséquent, elle est toujours finie grâce à l'hypothèse de bonne fondation de $\leq$. Autrement dit, tout nœud n'a qu'un nombre fini de fils.

(2) À chaque étape de la construction de l'arbre, toutes les branches qui partent de la racine contiennent dans l'ordre une suite d'états $q_1, \dots, q_n$ qui a la propriété qu'aucuns indices $i < j$ ne vérifient $q_i \trianglelefteq q_j$, sauf peut-être en bout de branche lorsque $j = n$, au moment d'un retour arrière. Ce n'est pas incompatible avec le fait que $\trianglelefteq$ soit un bon préordre parce que l'arbre courant est fini, ainsi que toutes ses branches.

[1] On peut aussi donner un preuve directe de la terminaison de cet algorithme, basée sur un argument diagonal, mais elle ne fait rien d'autre que de redémontrer de manière détournée le lemme de König.

Supposons que cet algorithme ne se termine pas. L'arbre que nous construisons est alors infini puisque qu'il croit à chaque itération de l'algorithme. Or tout nœud n'a qu'un nombre fini de fils (pt. 1). Grâce au lemme de König, on sait qu'il existe un chemin (une branche) infini dans l'arbre. Or les états $(q_n)_{n\in\mathbb{N}}$ qui étiquettent cette branche sont tels qu'aucuns indices $i < j$ ne vérifient $q_i \trianglelefteq q_j$ (pt. 2). Cela contredit l'hypothèse que $\trianglelefteq$ est un bon préordre. Donc l'arbre est fini et l'algorithme se termine. $\square$

La démonstration établit en fait un résultat plus général :

- Lorsque l'on atteint un point $q_i \trianglelefteq q_n$, le redémarrage peut avoir lieu à n'importe quelle profondeur $j \in [1..n]$, pas simplement en i. Il suffit pour cela de construire l'arbre comme si l'on avait eu $q_j \trianglelefteq q_n$.
- La valeur de redémarrage q_i' peut dépendre de tout l'historique $h = \langle q_1, \ldots, q_i, \ldots, q_n\rangle$; il suffit simplement que l'on ait la décroissance $q_i \leq \gamma(q_i, h)$.

Nous nous intéressons à présent à la nature de la fonction γ.

6.1.3 Fonction de généralisation.

Le choix de la fonction de généralisation $\gamma : E \times E \to E$ est un élément important de l'algorithme de redémarrage généralisé, non pas pour la garantie de terminaison, mais pour la qualité du résultat final obtenu à l'arrêt des transformations.

Tous les algorithmes de redémarrage généralisé que nous connaissont [Tur88, Has88, Sah91a, WCRS91] employent l'*anti-unification* comme fonction de généralisation, sans réelle justification ; c'est une heuristique établie, bien qu'elle ne soit pas déclarée comme telle. Nous tentons de l'expliquer.

Intuitivement, plus un terme est instancié, meilleur est son programme transformé (en comparaison au programme initial). Si l'on reprend l'exemple de la section §6.1.1, une définition de $\mathbf{facc}(n, 1, 1)$ qui ne laisse que n comme unique argument est potentiellement meilleure qu'une définition de $\mathbf{facc}(n, k, k)$ qui ne comporte que deux arguments : elle n'a pas à traiter toutes les valeurs possibles de k ; elle peut optimiser davantage le calcul lorsque $k = 1$. On peut tenir le même raisonnement avec les configurations $\mathbf{facc}(n, k, k)$ et $\mathbf{facc}(n, i, r)$.

Néanmoins, si l'on veut assurer la terminaison (prop. 6.1), il faut garantir qu'en chaque point de redémarrage q_i, on reprend les transformations avec comme base un terme q_i' plus général. C'était le cas avec $q_i = \mathbf{facc}(n, 1, 1) \geqslant \mathbf{facc}(n, k, k) = q_i'$.

D'autre part, il est raisonnable de penser — et c'est là qu'est l'heuristique — que ce qu'il y avait de commun entre les deux termes q_i et q_n a une chance de le rester dans des réductions ultérieures et mérite d'être conservé dans q_i' : ici, le fait que « $i = r$ » dans les termes $\mathbf{facc}(n, 1, 1)$ et $\mathbf{facc}(n, 2, 2)$ est préservé dans la configuration $\mathbf{facc}(n, k, k)$.

Cela a pour but d'éviter des redémarrages inutiles. En effet, toutes les généralisations de q_i sont a priori possibles. Dans le cas des schémas de programme, choisir par exemple $q_i' = \varphi(x, a)$ comme terme de redémarrage, quand $q_i = \varphi(f(g(x)), a)$ et $q_n = \varphi(h(x), a)$, c'est se dire ceci.

- Le terme plus général $\varphi(f(x), a)$ a peu de chance d'aboutir car la construction f semble consommée par la transformation.
- Donc la transformation de $\varphi(f(x), a)$ va probablement faire apparaître à nouveau le terme $\varphi(h(x), a)$.
- Par conséquent, il y aura aussi à nouveau redémarrage au même rang i sur une configuration plus générale.
- Pour éviter de faire des transformations sur $\varphi(f(x), a)$, qui seront oubliées lors du redémarrage qui suit, on préfère directement faire le redémarrage sur $\varphi(x, a)$.

Ce raisonnement s'étend aux schémas réguliers en considérant la généralisation sur les variables du langage et non sur les variables de schéma.

C'est pourquoi les algorithmes de redémarrage généralisé recherchent un *plus grand dénominateur commun*. En termes algébriques, il s'agit de l'*anti-unification* [Plo70]. Autrement dit, la fonction γ vérifie pour tout $t_1, t_2 \in M(F, X)$:

$$\gamma(t_1, t_2) \leqslant t_1, t_2 \quad \text{et} \quad \text{pour tout } t' \in M(F, X) \quad t' \leqslant t_1, t_2 \Rightarrow t' \leqslant \gamma(t_1, t_2)$$

Ainsi, dans le calcul $q'_i = \gamma(q_i, q_n)$, les contraintes sur q'_i ont le sens suivants.

- $q'_i \leq q_i$: garantir la terminaison de l'algorithme,
- $q'_i \leq q_n$: éviter des redémarrages inutiles,
- choisir un q'_i le plus particulier possible : permettre le plus d'optimisations.

C'est ce que réalise l'anti-unification.

La généralisation dans Fuse [WCRS91] fait également intervenir des sortes de types, qui représentent un ensemble de valeurs. Elles apportent un degré de finesse intermédiaire dans la généralisation, entre une valeur statique, connue, et une variable, qui peut désigner n'importe quelle valeur.

Il faut noter que l'état courant est en pratique un sous-terme du terme que l'on transforme. C'est le cas notamment lors d'appels emboîtés. Il est alors qualifié d'*actif* (active call) [Tur88, WCRS91]. En pratique, ce sous-terme est celui sur lequel porte la transformation courante ou suivante. C'est sur ce sous-terme que s'exprime le test du critère et la généralisation.

Le problème de trouver une bonne généralisation est aussi présent dans la preuve par récurrence. Par exemple, pour démontrer l'identité « $x + (x + x) = (x + x) + x$ » sur les entiers naturels, avec les axiomes ordinaires de l'addition, la récurrence sur x n'aboutit pas ; il faut d'abord généraliser la proposition en « $x + (y + z) = (x + y) + z$ ». Un système de démonstration automatique est un autre exemple d'application de l'algorithme de redémarrage généralisé : d'une part il contrôle la terminaison, et d'autre part il énumère par anti-unification des généralisations « intéressantes ».

6.2 Terminaison et critère d'arrêt.

Les problèmes de terminaison sont généralement indécidables, même dans des cas apparemment simples [Dau89, DLR93] : on ne peut pas toujours assurer que l'exécution va s'arrêter en un nombre fini d'étapes. Si l'on veut mettre en œuvre en toute sécurité un algorithme de transformation de programmes, il faut donc prendre des mesures coercitives.

On emploie pour cela un *critère d'arrêt*, qui procure la garantie qu'une exécution parcourt toujours un nombre fini d'étapes, quitte à l'interrompre prématurément. En effet, si l'on progresse par transformations *élémentaires* successives (cf. §1.1.5, transformations), un processus de transformations *composées* (cf. §1.1.5, algorithme) peut fournir malgré tout un résultat partiel lorsqu'il est interrompu.

Il diffère en cela des outils employés en réécriture qui fournissent une garantie de terminaison *a priori* [Der87] : on montre que le système de réécriture se termine, et plus aucun contrôle n'est nécessaire ensuite à l'exécution. En revanche, le problème est différent dans le cas des transformations de programme car l'enchaînement des transformations ne se termine pas en général ; c'est le critère d'arrêt qui aide à forcer la terminaison.

Nous avions remarqué (cf. §3.4, notes de bas de page numéros 7 et 9) que des limites physiques spatiales (la taille mémoire de la machine) font des coûts moyen et presque partout des exercices paradoxaux et dangereux. On relève un phénomène analogue à propos des critères d'arrêt : une limite physique temporelle (le temps que peut attendre un humain devant une machine, ou le temps que peut fonctionner une machine sans panne irrémédiable) bornent le nombre d'étapes pratique d'un processus d'exécution. Il vaut mieux obtenir un resultat partiel en un temps raisonnable, qu'un résultat meilleur en un temps qui, bien que toujours fini, est astronomique.

Il ne faut pas négliger non plus les cas de décidabilité spécifiques (par exemple [Dev90]) : de même qu'il est « raisonnable de chercher des améliorations optimales pour des classes de programmes relativement simples, et de se contenter de modestes mieux pour les classes plus complexes » (introduction du chapitre 4), il est aussi souhaitable d'inclure dans les systèmes automatiques de transformations de programme le maximum de cas décidables. C'est déjà ce qui se passe pour les programmes de manipulation de systèmes de réécriture, qui implémentent souvent un ensemble hétéroclite de techniques afin de couvrir plus ou moins automatiquement le maximum de cas courants.

Dans cette dernière section, nous n'apportons pas réellement de résultat nouveau ou novateur. Nous présentons simplement une démarche intellectuelle pour décider quel critère d'arrêt employer dans un système de transformation : définition, nature, qualités. En particulier, nous ne proposons pas une taxinomie des critères d'arrêt, comme nous l'avions fait au chapitre 3 pour les relations de coût. La section se conclut sur la suggestion d'un « bon » critère.

6.2.1 Critère d'arrêt.

Un *critère d'arrêt* est un test « pessimiste » que l'on effectue à chaque *étape* d'un processus itératif afin de déterminer s'il y a un *risque de non-terminaison*. Il prend en entrée un *état*, représentant l'étape courante, et un *historique* contenant la liste des états qui le précèdent. Il fournit en sortie l'une des réponses suivantes :

Pas de risque de boucle. On garantit que pour l'instant on est en terrain sûr et que l'on pourra plus tard interrompre l'exécution si le risque de ne pas s'arrêter devient trop grand.

Risque de boucle. On a détecté une possibilité de boucle infinie. Peut-être le processus va-t-il se terminer de lui-même un peu plus tard, mais ce ne peut être garanti. Il est conseillé d'interrompre l'exécution.

Un critère d'arrêt peut *éventuellement* affiner son verdict ainsi :

Boucle infinie certaine. Le critère d'arrêt certifie que l'on est entré dans une voie qui n'a pas de fin. Il faut interrompre l'exécution.

Pas de boucle. Si l'état courant permet d'anticiper la terminaison , on peut en toute sécurité poursuivre l'exécution comme dans le cas *pas de risque de boucle*. Il n'est même plus nécessaire d'avoir à nouveau recours au critère d'arrêt pour les étapes qu'il reste encore à parcourir.

Ces deux derniers jugements ne sont pas essentiels ; les réponses *risque de boucle* et *pas de risque de boucle* suffisent à constituer un critère d'arrêt.

D'autre part, un critère qui fait ces précisions a nécessairement une connaissance sur le mécanisme de fonctionnement du processus d'exécution. Sinon, déclarer une exécution infinie (resp. finie) en n'en examinant qu'une partie finie ne fait pas de sens.

À titre d'exemple, si l'on déclare une *boucle infinie certaine*, ce peut être parce que l'on repasse par un état strictement identique à un état antérieur comme le fait la fonction « `fun f(x) = 1::f(x)` ». Plus généralement, cela peut être dû à un état qui a grossi *inconditionnellement*. C'est le cas par exemple pour la fonction « `fun f(x) = 1::f(2::x)` » dont l'évaluation boucle à l'infini en construisant au dessus d'un terme sans jamais regarder sa structure. De manière plus générale encore, c'est aussi le cas pour les fonctions comme « `fun f(2::x) = 1::f(2::3::x)` » qui peuvent explorer l'argument (ici par le motif `2::x`) mais qui reprennent l'exécution en construisant un nouveau terme de même frontière extérieure (ici `2::3::x`).

6.2.2 Nature des critères.

Nous avons déjà rencontré quelques types de critères d'arrêt (cf. §1.3.4, 1.4.3, terminaison). Les constructions les plus courantes en évaluation partielle sont les suivantes :

Décroissance. Une fonction entière τ définie sur les états x_n doit décroître strictement à chaque étape : $\tau(x_{n-1}) > \tau(x_n)$. Si c'est le cas, on répond *pas de risque de boucle*, sinon la réponse est *risque de boucle*. C'est par exemple la fonction taille $|\,.\,|$ sur des états qui sont des termes algébriques. Dans l'historique, on ne conserve en pratique que l'état précédent x_{n-1}, ou simplement l'entier $\tau(x_{n-1})$. Plus géneralement, ce type de critère exprime que la suite des états doit décroître strictement selon une relation bien fondée.

Borne. Une fonction entière τ doit rester dans les limites d'une borne b, qui peut être fixée, arbitraire, ou bien déterminée par $\tau(x_0)$. Cette fonction doit en outre être telle que le nombre d'états x ayant une image $\tau(x)$ donnée doit être fini, à la manière de la taille définie pour le coût asymptotique (cf. §3.4.9). Tant que l'image par τ de l'état courant ne dépasse pas cette borne et que les états parcourus diffèrent, c'est-à-dire si $\tau(x_n) \leqslant b$ et $\forall i < n \quad x_i \neq x_n$, on répond *pas de risque de boucle*, sinon la réponse est *risque de boucle*. Dans une formulation plus générale, on quotiente les états par une relation d'équivalence d'index fini et l'on teste si $x_i \approx x_n$. L'historique est ici constitué de l'ensemble des états parcourus. Là aussi, τ peut être la fonction taille $|\,.\,|$ sur les termes.

Beau préordre. Le critère d'arrêt est défini par un beau préordre $\trianglelefteq$ (cf. §6.1) et l'historique composé des états déjà parcourus. Si l'on trouve à l'étape n un indice $i < n$ tel que $x_i \trianglelefteq x_n$, la réponse est *risque de boucle*, sinon elle est *pas de risque de boucle*. Comme dans le cas de la décroissance à propos de la relation bien fondée, ce beau préordre est défini en pratique par l'application d'une *fonction de transport* τ qui envoie les états vers un beau préordre connu.

On peut remarquer que le type *borne* est un cas particulier du type *décroissance* ; c'est le nombre d'états non encore parcourus et inférieurs à la borne fixée qui décroît. De même, le type *borne* est un cas particulier du type *beau préordre* où la relation est définie par $x_i \trianglelefteq x_n$ ssi $b < \tau(x_n)$.

Par ailleurs, les démonstrations qui prouvent qu'une relation est un *beau préordre* font parfois intervenir une *décroissance*, car ce sont des preuves par récurrence[2]. Il existe néanmoins des preuves non-constructives, qui n'admettent pas d'équivalent apparent [NW63].

[2] On peut également noter que le rapport entre un historique réduit à l'état précédent, et un historique composé de tous les états déjà parcourus, est du même ordre qu'entre la récurrence faible (la proposition $P(n+1)$ est déduite de $P(n)$ seul) et la récurrence forte ($P(n+1)$ est déduite de $P(n), P(n-1), \ldots, P(0)$).

6.2.3 Qualités d'un critère d'arrêt.

De la même manière que nous avions examiné les *qualités* d'une relation de coût statique (les propriétés souhaitables, cf. §3.5), nous examinons les *qualités d'un critère d'arrêt*. En revanche, nous ne développons pas une formulation aussi formelle car beaucoup de notions sont spécifiques au processus d'exécution considéré ou au mode de construction du critère.

Permissivité. La qualité première d'un critère d'arrêt est d'être *permissif* (tout en restant *sûr*) ; il doit permettre de poursuivre les exécutions autant que possible. Dans le cadre de l'évaluation partielle, aller plus loin signifie optimiser davantage. À l'opposé, un mauvais critère d'arrêt est trop strict ; il voit des *risques de boucle* qui n'en sont pas et interrompt prématurément les exécutions. Dans la classification des *tests de boucle*[3] en programmation logique de [BAK91], être plus permissif, c'est être plus *fort* (stronger).

Promptitude. Un bon critère d'arrêt est aussi un critère *prompt*, qui agit au plus tôt : lorsqu'une réelle boucle infinie « survient », il doit la détecter rapidement (*risque de boucle* ou éventuellement *boucle infinie certaine*) et ne pas laisser l'exécution continuer pour l'interrompre, finalement, un peu plus tard. En pratique, dans le cas d'un processus d'optimisation, un critère moins prompt déroule inutilement des boucles, de sorte que le code généré est plus volumineux pour une performance similaire. Par ailleurs, le processus d'optimisation lui-même est plus coûteux en temps et en espace.

Perspicacité. Un bon critère est également *perspicace* ; il déclare *boucle infinie certaine* là ou un autre dit simplement *risque de boucle*. Dans certains cas, cette information peut être la source d'optimisation [Bol91].

En revanche, déclarer *pas de boucle*, puis finir quelques étapes plus loin, là où un autre critère dit simplement *pas de risque de boucle* et s'arrête, de la même manière, après ces mêmes étapes, n'a pas d'intérêt majeur, si ce n'est une économie du nombre de tests.

Critère intrinsèque. Un bon critère d'arrêt est un critère *intrinsèque*, par opposition à un critère qui repose sur la donnée extérieure de paramètres, comme des bornes à certains compteurs (cf. §1.3.4, terminaison). Par ailleurs, ajuster des valeurs en vue d'obtenir un bon résultat nécessite une connaissance sur le mécanisme d'optimisation. Néanmoins, bien que cela entache aussi le caractère automatique d'un système de transformation, cette paramétrisation peut parfois être justifiée si elle permet de mieux traiter un problème plus spécifique.

Coût d'évaluation. Enfin, il ne faut pas non plus perdre de vue l'aspect pratique. Un critère est bon si son évaluation est peu coûteuse. Bien que ce dernier point n'influe pas en théorie sur la nature du résultat, il peut le faire en pratique si les temps de calcul sont très longs (voir le paradoxe cité dans l'introduction de la section §6.2).

À titre d'exemple, nous indiquons comment s'expriment ces qualités sur les critères explicités à la section §6.2.2, dont nous reprenons aussi les notations.

- Un critère *décroissance*$_1$ est plus permissif qu'un critère *décroissance*$_2$ si $\tau_1 \geqslant \tau_2$
  Un critère *borne*$_1$ est plus permissif qu'un critère *borne*$_2$ si $b_1 \geqslant b_2$ et $\tau_1 \leqslant \tau_2$
  Un critère *préordre*$_1$ est plus permissif qu'un critère *préordre*$_2$ si $\trianglelefteq_1 \subset \trianglelefteq_2$
- Un critère *décroissance*$_1$ est plus prompt qu'un critère *décroissance*$_2$ si $\tau_1 \leqslant \tau_2$
  Un critère *borne*$_1$ est plus prompt qu'un critère *borne*$_2$ si $b_1 \leqslant b_2$ ou $\tau_1 \geqslant \tau_2$
  Un critère *préordre*$_1$ est plus prompt qu'un critère *préordre*$_2$ si $\trianglelefteq_1 \supset \trianglelefteq_2$
- Les critères du type *décroissance* et *beau préordre* sont intrinsèques alors que les critères de type *borne* ne le sont pas.

[3]Ces tests sont employés pour corriger dynamiquement l'incomplétude de la règle de résolution de PROLOG.

- Il n'y a pas de propriété générale qui concerne la perspicacité car c'est une qualité qui suppose une connaissance spécifique sur le processus d'exécution.
- Si les coûts des tests sont voisins, un critère de type *décroissance* est moins coûteux qu'un critère de type *borne* ou *beau préordre* car la comparaison est relative au seul état précédent, et non à tout l'historique.

Il faut noter que les propriétés concernant la permissivité et la promptitude ne sont que des conditions *suffisantes*.

D'autre part, on remarque que permissivité et promptitude sont paradoxalement des qualités « opposées » car les conditions suffisantes mentionnées sont contraires l'une de l'autre. Cela vient du fait que la perspicacité fait du sens sur les exécutions qui se terminent, et la promptitude sur celles qui ne se terminent pas.

La permissivité n'est pas une qualité « difficile » à réaliser, mais c'est aux dépens de la promptitude et du caractère intrinsèque. En effet, il est toujours possible de fixer arbitrairement une profondeur d'exploration de l'exécution. Plus cette profondeur est grande, plus on est permissif. En revanche, on est d'« autant » moins prompt. Par ailleurs, un tel critère n'est pas intrinsèque.

6.2.4 Un bon critère d'arrêt.

Pour motiver ce bavardage, nous indiquons à présent un critère d'arrêt qui nous semble excellent au sens des qualités définies à la section précédente (cf. §6.2.3). D'une part, il est intrinsèque. D'autre part, il réalise un bon compromis entre permissivité et promptitude. Enfin, son coût d'évaluation reste (presque) raisonnable. Il est défini à l'aide de la relation suivante.

Définition 6.2. (Plongement homéomorphe pur)
Soit F une signature. La relation $\trianglelefteq$ de *plongement homéomorphe pur* (pure homeomorphic embedding) dans $M(F)$ est définie par :

- $\forall s, t \in M(F) \quad s = f(s_1, \dots, s_m) \trianglelefteq g(t_1, \dots, t_n) = t$ ssi
  $(\exists i \in [n] \quad s \trianglelefteq t_i)$ ou $(f = g$ et $\forall j \in [n] \quad \exists i_1, \dots, i_m \in \mathrm{IN} \quad 1 \leqslant i_1 < \cdots < i_m \leqslant n$ et $s_j \trianglelefteq t_{i_j})$

Il s'agit en fait du plongement homéomorphe de la relation d'égalité (cf. §N.10). C'est une relation d'ordre.

Le critère s'exprime ainsi.

Définition 6.3. (Critère de Higman-Kruskal)
Si F est une signature finie, dont les opérateurs ne sont pas nécessairement d'arité fixe, la relation $\trianglelefteq$ de plongement homéomorphe dans $M(F)$ est un bel ordre. Elle définit donc un critère d'arrêt pour tout processus dont les états varient (naturellement ou par l'intermédiaire d'une fonction de transport, cf. §6.2.2) dans $M(F)$. Nous le désignerons sous le nom de *critère de Higman-Kruskal*.

En effet, on doit à Higman le cas d'une signature F finie dont les opérateurs sont d'arité fixe [Hig52], et à Kruskal la généralisation à une arité quelconque [Kru60]. La section §N.10 donne une formulation plus complète du lemme de Higman et du théorème de Kruskal, dont la proposition précédente est dérivée.

Les plongements qui font l'objet de ce lemme et de ce théorème sont fréquemment utilisés pour résoudre des problèmes de terminaison dans les systèmes de réécriture [Der87, DJ90]. En particulier, la notion d'*ordre de simplification* (simplification ordering) est très proche de celle du plongement ; il a d'ailleurs été démontré que si deux termes sans variables s et t sont tels que $s \trianglelefteq t$, alors t ne peut être inférieur à s dans aucun ordre de simplification [Der82]. Néanmoins, parce que les objectifs diffèrent (cf. l'introduction de la section §6.2), le critère même est d'un usage plus rare. Il a toutefois été utilisé dans un but voisin par Plaisted dans le cadre de la procédure de complétion de Knuth-Bendix [Pla86]. Par ailleurs, nous prétendons que le critère de terminaison employé dans l'algorithme de supercompilation de REFAL est un cas particulier du critère de Higman (cf. §1.4.3, terminaison). Il apparaît également dans [BAK91], que nous citions plus haut à propos du test de boucle.

Afin de justifier notre intérêt pour ce critère, nous examinons quelles qualités (au sens de la section §6.2.3) il vérifie. Pour les remarques qui suivent, nous aurons besoin de la proposition élémentaire suivante, qui exprime la monotonie de la fonction taille par rapport aux relations $\trianglelefteq$ et $\leqslant$.

Proposition 6.2. (Propriété du plongement homéomorphe pur)
La relation $\trianglelefteq$ de plongement homéomorphe pur vérifie :

- $\forall s, t \in M(F)$ si $s \trianglelefteq t$ alors $|s| \leqslant |t|$

Par conséquent, si t est un sous-terme propre de s, alors $s \ntrianglelefteq t$.

Pour motiver l'étude de ce critère particulier, notons enfin que sa formulation algébrique permet un emploi facile et immédiat dans un grand nombre de circonstances : systèmes de transformation, langages et schémas de programmes. . .

Critère intrinsèque.

La première remarque que l'on peut faire sur ce critère d'arrêt est qu'il est intrinsèque ; il ne nécessite pas la donnée de paramètres de contrôle ; il suffit de savoir que la signature est finie. Il arrive toutefois que des signatures soient infinies. Nous examinons quelques cas :

Infinitude des entiers. Si des données en nombre infini, comme les entiers par exemple, apparaissent explicitement dans la signature, il faut leur substituer une représentation qui emploie une signature finie. Ainsi, les entiers positifs peuvent être remplacés pour les besoins du test par une représentation sous forme unaire : $\{\mathsf{0} : \mathrm{nat}\ ,\ \mathsf{s} : \mathrm{nat} \rightarrow \mathrm{nat}\}$, ou plus généralement dans une base quelconque.

En pratique, il est plus simple de considérer en fait le plongement homéomorphe (cf. §N.10) non pas de l'égalité sur tous les symboles, mais du bel ordre défini comme « $\leqslant$ » sur les entiers et « $=$ » sur les autres symboles. On évite ainsi une représentation intermédaire inutile.

En ce qui concerne les entiers négatifs, on peut par exemple les faire intervenir par l'intermédiaire de leur valeur absolue. Il faut toutefois noter que ces adaptations particularisent le critère avec des opérations centrées autour de zéro (ce serait tout autant le cas avec un entier autre que zéro). En toute rigueur, il n'est pas réellement intrinsèque.

Si ce caractère intrinsèque n'est pas important, on peut aussi quotienter un ensemble infini d'opérateurs par une classe d'équivalence d'index fini (cf. §1.3.4, terminaison).

Infinitude pour cause de typage. L'infinitude de la signature peut être due à des raisons de typage. C'était le cas par exemple pour nos transformations de schémas algébriques en schémas réguliers (cf. §2.5.8, 2.5.9), où il fallait introduire un opérateur $\mathsf{call}_{s_1 \ldots s_n s}$ pour chaque type $s_1 \times \cdots \times s_n \rightarrow s$. Dans l'exemple de BOOL, la signature, finie dans la spécification algébrique, devient ainsi infinie dans le cas régulier.

Pour lever ce problème, il suffit d'omettre l'information de typage et de ne conserver, au moment du test, qu'un unique opérateur call d'arité quelconque.

Infinitude des variables. Que les variables soient algébriques (dans le cas des schémas de programme algébriques) ou syntaxiques (dans le cas des schémas de programme réguliers), elles se présentent toujours en nombre infini. Le dépliage notamment peut en énumérer une infinité. C'est le cas également des variables de la programmation logique, qui peuvent être fabriquées dynamiquement à l'exécution.

Il faut noter que pour le plongement homéomorphe pur, les variables font partie de la signature au même titre que les autres opérateurs. Il ne s'agit pas d'un filtrage (voir pour cela [Sti88]).

Une première solution est de confondre toutes les variables ; ainsi, un terme comme $\mathsf{if}(x, \mathsf{true}, \varphi(y, x))$ ou $\mathsf{if}(\mathsf{var}(x), \mathsf{true}, \varphi(\mathsf{var}(y), \mathsf{var}(x)))$ est traduit en $\mathsf{if}(\mathsf{var}, \mathsf{true}, \varphi(\mathsf{var}, \mathsf{var}))$. Néanmoins, cette simplification est un peu grossière au sens de la permissivité ; elle interrompt en particulier une dérivation $\varphi(x, y) \to^* \varphi(z, z)$ alors que l'on peut imaginer que la « vraie » boucle se situe en fait au niveau d'un $\varphi(z, z) \to^* \varphi(z', z')$. De même, elle interrompt une permutation $\varphi(x, y, z) \to^* \varphi(y, z, x) \to^* \varphi(z, x, y) \to^* \varphi(x, y, z)$ au tout début, sans couvrir un cycle complet et attendre la boucle « réelle ».

Lorsque l'ensemble des variables est dénombrable, on peut associer un entier à chaque variable. On se ramene ainsi aux cas ci-dessus de la représentation des entiers par une signature finie, ou de l'adaptation du plongement homéomorphe. Cependant, le critère n'est plus intrinsèque car il faut spécifier la fonction des variables vers les entiers. On peut éviter cela en numérotant les variables avec une notation normalisée « à la de Bruijn » : des termes comme $\mathsf{if}(x, \mathsf{true}, \varphi(y, x))$ ou $\mathsf{if}(\mathsf{var}(x), \mathsf{true}, \varphi(\mathsf{var}(y), \mathsf{var}(x)))$ sont alors traduits en $\mathsf{if}(\mathsf{var}(0), \mathsf{true}, \varphi(\mathsf{var}(1), \mathsf{var}(0)))$.

De plus, afin de tenir compte des variables communes — cela a surtout un sens en programmation logique — on peut attribuer les indices *simultanément* dans les deux termes à comparer. Ainsi, au cours de la permutation $\varphi(x, y, z) \to^* \varphi(y, z, x) \to^* \varphi(z, x, y) \to^* \varphi(x, y, z)$, les termes $\varphi(\mathsf{var}(0), \mathsf{var}(1), \mathsf{var}(2))$, $\varphi(\mathsf{var}(1), \mathsf{var}(2), \mathsf{var}(0))$ et $\varphi(\mathsf{var}(2), \mathsf{var}(0), \mathsf{var}(1))$ ne sont pas comparables entre eux, et l'on ne signale aucun de risque de boucle. Par contre, on s'arrête bien au bout d'une période exactement, lorsqu'on retombe sur le même terme : $\varphi(\mathsf{var}(0), \mathsf{var}(1), \mathsf{var}(2)) \trianglelefteq \varphi(\mathsf{var}(0), \mathsf{var}(1), \mathsf{var}(2))$. On réalise ainsi un bon compromis entre permissivité et promptitude.

Néanmoins, cette adaptation n'est que partiellement satisfaisante car elle introduit un ordre entre les variables qui, bien qu'il ne dépende pas d'une quantité extérieure, varie cependant en fonction du nombre de variables et de leur position dans le terme.

Si l'on veut conserver un critère intrinsèque, il faut examiner *toutes* les numérotations possibles des variables. Par exemple, pour le terme $t = \mathsf{if}(x, \mathsf{true}, \varphi(y, x))$, un test $s \trianglelefteq t$ correspondra à la disjonction des deux tests $s \trianglelefteq \mathsf{if}(\mathsf{var}(0), \mathsf{true}, \varphi(\mathsf{var}(1), \mathsf{var}(0)))$ et $s \trianglelefteq \mathsf{if}(\mathsf{var}(1), \mathsf{true}, \varphi(\mathsf{var}(0), \mathsf{var}(1)))$. Plus généralement, si s et t font apparaître n variables distinctes, le test $s \trianglelefteq t$ demande $n!$ tests élémentaires $s_i \trianglelefteq t_i$ correspondants aux différentes manière d'indicer les variables. On peut aussi voir cela comme les tests de plongement homéomorphes, non pas purs, mais construits sur les $n!$ relations d'ordre correspondant aux différentes manières de comparer les n variables.

Cette dernière formulation est intrinsèque et ne privilégie pas une manière de regarder un terme. Elle permet également de n'interrompre une permutation qu'en fin de période. En revanche, son évaluation est plus coûteuse (d'un facteur $n!$). Toutefois, il est possible de factoriser une grande partie des vérifications structurelles dans les $n!$ tests $s_i \trianglelefteq t_i$ pour se concentrer principalement sur les différentes manière d'indicer.

Environnements explicites. Dans le cas d'environnements explictes, il faut expliciter à l'aide de quelques opérateurs (un nombre fini) les manipulations de stokage et d'accès. Qui plus est, cela va dans le sens d'une représentation plus concrète, donc potentiellement plus fidèle, de l'éxécution (cf. §2.3).

Permissivité.

Comme nous l'avons dit plus haut, il n'y a pas de caractérisation générale de la permissivité. C'est sur des exemples que nous l'avons évoqué à propos de l'infinitude des variables, et c'est encore sur des exemples que nous l'illustrons ici.

Distributivité. Considérons l'application d'une règle de distributivité $\langle f(x, g(y,z)) \to g(f(x,y), f(x,z))\rangle$. Alors qu'un critère de décroissance basé sur la taille des termes dit *risque de boucle* car $|g(f(x,y), f(x,z))| = 3 + 2\,|x| + |y| + |z| > 2 + |x| + |y| + |y| = |f(x, g(y,z))|$, le critère $\trianglelefteq$ déclare *pas de risque de boucle*. En effet, $f(x, g(y,z)) \trianglelefteq g(f(x,y), f(x,z))$ ssi $f(x, g(y,z)) \trianglelefteq f(x,y)$ ou $f(x, g(y,z)) \trianglelefteq f(x,z)$. Or cela est impossible (prop. 6.2).

Fonction de Ackermann. Pour montrer que ce critère permet de poursuivre une exécution relativement loin, nous donnons l'exemple du contrôle de terminaison de la fonction de Ackermann, fonction non primitive récursive dont la croissance extrêmement rapide est notoire. Nous transformons pour cela la définition donnée à la section §1.3.2 afin d'employer des entiers en notation unaire.

```
fun ack(   O ,    y  ) = S(y)
  | ack( S(x),    O  ) = ack( x, S(O) )
  | ack( S(x), S(y) ) = ack( x, ack( S(x), y ))
```

Cette formulation, traduite en termes de règles de réécritures, est ainsi construite sur une signature finie :

$$\begin{array}{lcl} \mathbf{ack}(0, y) & \to & \mathsf{s}(y) \\ \mathbf{ack}(\mathsf{s}(x), 0) & \to & \mathbf{ack}(x, \mathsf{s}(0)) \\ \mathbf{ack}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathbf{ack}(x, \mathbf{ack}(\mathsf{s}(x), y)) \end{array}$$

Nous montrons que, pour tout $p, q \in \mathbb{N}$, aucune dérivation de $\mathbf{ack}(\mathsf{s}^p(0), \mathsf{s}^q(0))$ selon la stratégie de l'appel par valeur[4] ne vérifie la condition de plongement. Nous ne donnons que les grandes lignes de la démonstration.

1. On montre par récurrence sur la longueur de la dérivation que tout terme t qui apparaît dans la dérivation $\mathbf{ack}(\mathsf{s}^p(0), \mathsf{s}^q(0)) \to^* t$ est de la forme $\mathsf{s}^n(0)$ ou bien $\mathbf{ack}(\mathsf{s}^{k_1}(0), \ldots \mathbf{ack}(\mathsf{s}^{k_{n-1}}(0), \mathsf{s}^{k_n}(0)) \ldots)$ avec $n \geqslant 1$ et $k_1, \ldots, k_n \geqslant 0$. On emploie pour cela l'hypothèse sur la stratégie de réduction.

2. Du fait de la structure en peigne très particulière de ces termes, pour toute dérivation de la forme $\mathbf{ack}(\mathsf{s}^p(0), \mathsf{s}^q(0)) \to^* t \to^+ t'$, la relation $t \trianglelefteq t'$ se traduit de la manière suivante.

$$t = \mathbf{ack}(\mathsf{s}^{k_1}(0), \ldots \mathbf{ack}(\mathsf{s}^{k_{n-1}}(0), \mathsf{s}^{k_n}(0)) \ldots) \trianglelefteq \mathbf{ack}(\mathsf{s}^{k'_1}(0), \ldots \mathbf{ack}(\mathsf{s}^{k'_{n'-1}}(0), \mathsf{s}^{k'_{n'}}(0)) \ldots) = t'$$

$$\text{ssi}$$

$$\exists (i_j)_{j \in [n]} \quad 1 \leqslant i_1 < \cdots < i_n \leqslant n' \ \text{ tq } \ \forall j \in [n] \quad k_j \leqslant k'_{i_j}$$

[4] Le résultat reste vrai pour une stratégie quelconque, mais la démonstration est plus longue. Il l'est également pour une formulation en termes de schémas de programmes, par oppposition aux motifs que comportent la définition de **ack** dans la définition en termes de règles de réécritures ci-dessus.

Or, la fonction ack est *strictement* croissante en chacun de ses arguments, de même que les fontions $\lambda y.ack(c, y)$ pour $c \in \mathbb{N}$ fixé :

$$\begin{array}{rcl} ack(x,y) & > & y \\ ack(x+1,y) & > & ack(x,y) \\ ack(x,y+1) & > & ack(x,y) \end{array}$$

Dans la mesure où t et t' s'évaluent en la même valeur, il est donc impossible d'une part que $n' > n$, et d'autre part que $k_j < k'_{i_j}$ pour n'importe quel $j \in [n]$. Par conséquent, $t = t'$ ssi $n = n'$ et $\forall j \in [n] \quad k_j \leqslant k'_j$. Autrement dit, la seule possibilité d'avoir $t \trianglelefteq t'$ au cours d'une dérivation est que $t = t'$.

3. Puisque la stratégie est fixée, déterministe, cela signifie alors que la dérivation ne se termine pas car on repasse par un même terme. Or on sait que l'exécution de ack se termine, et en particulier pour cette stratégie de réduction. Il donc également impossible que $t = t'$.

4. Par conséquent, aucuns termes t et t' obtenus par $\mathbf{ack}(\mathsf{s}^p(0), \mathsf{s}^q(0)) \rightarrow^* t \rightarrow^+ t'$ ne vérifient $t \trianglelefteq t'$.

Le critère de Higman-Kruskal permet donc l'exécution complète de la fonction de Ackermann.

Promptitude.

Être *prompt*, c'est détecter les boucles au plus tôt. On peut remarquer que le test de plongement $\trianglelefteq$ est positif lorsque tous les arguments d'un appel récursif ont cru (ou sont restés constants). Cela vaut pour les entiers :

$$\text{si} \quad \text{pour tout } i \in [k] \quad n_i \leqslant n'_i \quad \text{alors} \quad f(\mathsf{s}^{n_1}(0), \ldots, \mathsf{s}^{n_k}(0)) \trianglelefteq f(\mathsf{s}^{n'_1}(0), \ldots, \mathsf{s}^{n'_k}(0))$$

et plus généralement pour toutes les structures de données, dès qu'il y a imbrication dans la construction des termes. Cela permet d'interrompre un grand nombre de boucles infinies parmi les plus courantes, juste avant la seconde itération.

Coût d'évaluation.

La définition du test de plongement fait apparaître une disjonction. L'évaluation du test semble être relativement coûteuse car il faut effectuer récursivement un branchement sur les deux cas. Néanmoins, en dépit des apparences, le test de plongement n'est pas exponentiel mais linéaire [Sti88, Gus92] ; il est proportionnel à la taille des deux termes testés et à la plus grande arité.

En revanche, si l'on désire prendre en compte au niveau du test des propriétés liées à la sémantique du langage, comme l'associativité ou la commutativité, la complexité augmente. À titre de référence, nous avons extrait de [Sti88] les résultats mentionnés dans le tableau 6.1.

Des heuristiques aident également une mise en œuvre pratique. La contraposée de la proposition 6.2 fournit à faible coût une condition suffisante de non comparaison. D'autre part, si l'on prend aussi en compte dans l'algorithme les types des fonctions, on élimine les comparaisons d'opérateurs qui ne font pas de sens.

Une voie de recherche consiste à trouver une condition de plongement plus faible, qui reste compatible avec le lemme de Higman ou le théorème de Kruskal, et dont l'évaluation soit moins coûteuse. En effet, la

Type de plongement	Complexité
homéomorphe pur	$\mathcal{O}(\|s\| \,.\, \|t\| \,.\, a)$
avec opérateurs associatifs	$\mathcal{O}(\|s\|^2 .\, \|t\| \,.\, a)$
avec opérateurs commutatifs	$\mathcal{O}(\|s\| \,.\, \|t\| \,.\, a^{2,5})$
avec opérateurs associatifs et commutatifs	problème NP-complet

(le nombre a est la plus grande arité qui apparaît dans s et t)

Tableau 6.1 : Complexité du test de plongement $s \trianglelefteq t$

suite de termes énumérée lors d'une exécution n'est pas quelconque ; chaque terme t_{i+1} est déduit de t_i par une opération élémentaire qui n'explore en pratique qu'une partie du terme t_i.

Par exemple, considérons des constructeurs de types de données f, g, h et h'. Si $\varphi(f(h(g(h'(a)))))$ suit immédiatement $\varphi(f(g(a)))$ — il y a alors plongement — cela ne peut être dû qu'à des règles qui, en terme de motifs, représentent $\langle\varphi(f(g(a))) \to \varphi(f(h(g(h'(a)))))\rangle$, $\langle\varphi(f(g(x))) \to \varphi(f(h(g(h'(a)))))\rangle$, $\langle\varphi(f(x)) \to \varphi(f(h(g(h'(a)))))\rangle$, ou $\langle\varphi(x) \to \varphi(f(h(g(h'(a)))))\rangle$. Or ces règles ne permettent pas de boucler ou bouclent trivialement. En revanche, si $\varphi(f(h(g(h'(a)))))$ ne suit pas immédiatement $\varphi(f(g(a)))$, cela signifie que l'insertion de h et de h' s'est effectuée en deux étapes (sinon, on peut reproduire le même raisonnement que précédemment). Par conséquent, on peut imaginer limiter la profondeur du test d'imbrication de la définition plongement et réduire ainsi la combinatoire.

Conclusion du chapitre.

Nous avons donné une formalisation abstraite de l'algorithme de redémarrage généralisé (déf. 6.1) et ouvert ainsi son champ d'application. Après avoir dit en quoi sa terminaison n'avait pas été rigoureusement prouvée, nous en avons donné une démonstration complète (prop. 6.1). Puis, nous avons étudié la nature de la fonction de généralisation, et dit notamment en quoi l'usage courant de l'anti-unification était une heuristique (cf. §6.1.3).

D'autre part, nous avons étudié la notion de critère d'arrêt sous différents angles : réponses produites (cf. §6.2.1), nature (cf. §6.2.2) et qualités (cf. §6.2.3). Cette analyse a permis de justifier les mérites du critère de Higman-Kruskal. En particulier, nous avons montré qu'il pouvait contrôler l'évaluation totale de la fonction de Ackermann (cf. §6.2.4, permissivité).

Là s'arrête notre discours. Nous dressons à présent un bilan de notre parcours et concluons sur les limites de l'optimisation par évaluation partielle.

Conclusion

Avant de clore cette étude, nous faisons une récapitulation rapide des points abordés dans le document. Le lecteur trouvera un compte rendu plus détaillé, comportant notamment la référence des résultats obtenus, en conclusion de chacun des chapitres. Nous mentionnons ensuite quelques voies de recherche qui poursuivent ces travaux, et concluons sur les limites de l'évaluation partielle.

Chemin parcouru.

Nous retraçons ici les étapes importances de cette thèse. Elle a débuté sur un panorama de l'évaluation partielle.

1. **Tour d'horizon de l'évaluation partielle.** Nous avons présenté dans le premier chapitre une vision analytique et synthétique de l'évaluation partielle, centrée sur les trois grands axes : spécialisation, supercompilation, évaluation partielle généralisée.

   Nous avons mis en évidence quelques déficiences qui, sans remettre en cause les applications pratiques, montrent néanmoins l'insuffisance de certaines justifications. Elles concernent notamment la nature des optimisations et la terminaison des algorithmes.

Une seconde partie s'est consacrée à la formalisation du problème de l'évaluation partielle.

2. **Sémantique et exécution.** Afin de poser des bases pour le raisonnement, nous avons proposé, et illustré à l'aide d'un exemple de compilation, une formalisation des modèles d'exécution. Nous avons indiqué à quels niveaux s'établissaient les compromis entre le degré d'abstraction d'un modèle d'exécution et sa fidélité à une implémentation. Nous avons montré comment interpréter comme modèle d'exécution une spécification en sémantique naturelle ou au moyen de schémas de programmes.

   Par ailleurs, nous avons étendu le type de règles employé par la sémantique opérationnelle des schémas de programmes afin de mieux pouvoir représenter un modèle d'exécution donné. Nous avons également montré les avantages d'un schéma régulier sur un schéma algébrique en termes de puissance d'expression et de fidélité, et fourni des procédés de traduction de l'un à l'autre.

3. **Mesure de performance.** Ensuite, nous avons formalisé un modèle de performance à l'aide d'une notion de coût abstrait, capable de représenter des quantités comme le temps d'exécution ou l'espace mémoire consommé. Nous avons défini une mesure de performance générique sur les modèles d'exécution spécifiés au moyen de la sémantique naturelle ou des schémas de programme. Nous avons ainsi montré comment attribuer un coût à une exécution, puis à un programme, considéré comme un ensemble d'exécutions possibles.

   Ce cadre formel nous a permis de construire ensuite toute une gamme de comparaisons de coût (coût absolu, moyen, maximal, presque partout, fini, presque fini, limite, asymptotique, majoré, selon une distribution). Nous avons étudié quelles qualités rechercher dans une comparaison de coût, comment ces qualités étaient reliées entre elles, et lesquelles étaient vérifiées par les coûts que nous avions construits. Enfin, nous avons établi les propriétés d'inclusion de ces modèles de coûts afin de fournir des échelles de comparaison sur les transformations.

4. **Optimisation et optimalité.** Cette étude a conduit à une définition formelle de l'évaluation partielle. Nous avons montré l'existence de trois types de « meilleures évaluations partielles » (optimale forte, optimale faible et minimale), que nous avons illustrés sur les modèles de coût précédents. L'examen de leurs propriétés générales a fourni des renseignements d'ordres qualitatif et quantitatif qui permettent de mieux apprécier un résultat d'évaluation partielle.

   Nous avons ensuite analysé les difficultés intrinsèques des évaluations partielles optimales et minimale. Nous avons donné des méthodologies pour affirmer ou infirmer l'existence de ces meilleures évaluations partielles, et montré qu'en général il n'en existait pas. Toutefois, dans le cas d'un programme de domaine fini, et moyennant quelques hypothèses sur le modèle de performance, nous avons pu établir qu'il existait toujours une évaluation partielle minimale. D'autre part, nous avons montré et illustré comment employer un langage intermédiaire pour résoudre des problèmes d'évaluation partielle optimale ou minimale.

Une troisième partie a présenté des outils pour réaliser un système automatique de transformation de programme.

5. **Transformations et correction.** Nous nous sommes intéressé à la correction des transformations de pliage/dépliage dans le cadre des schémas de programmes récursifs. Nous avons repris la condition de dépliage/pliage faible et établi des lemmes de commutativité sur les systèmes de réécriture. Nous avons ainsi pu étendre l'hypothèse de linéarité à gauche *et* à droite des règles sémantiques en une condition plus faible de linéarité à gauche *ou* à droite. Nous pouvons ainsi statuer sur la correction dans davantage de cas pratiques.

   Nous avons également examiné le cas d'une séquence de transformations qui autorise plus de liberté dans le choix des schémas employés pour faire les pliages et les dépliages. Nous avons montré comment établir sa correction en la ramenant au cas précédent.

6. **Algorithme et terminaison.** Nous avons ensuite analysé l'algorithme de redémarrage généralisé. Nous en avons donné une formulation abstraite et nous avons expliqué pourquoi les preuves de sa terminaison étaient hâtives. Nous avons montré que l'algorithme terminait néanmoins, sans hypothèses supplémentaires. Par ailleurs, nous avons aussi étudié la nature de la fonction de généralisation.

   Nous avons également fait l'étude des critères d'arrêts, outils de contrôle de l'algorithme de redémarrage généralisé et, plus généralement, des algorithmes d'évaluation partielle. Nous avons examiné leur nature et énoncé quelles étaient leurs « bonnes » propriétés. Nous avons conseillé et justifié l'emploi d'un critère d'arrêt particulier, fondé sur le théorème de Kruskal.

Nous donnons à présent quelques prolongements possibles de cette approche.

Perspectives.

Nous voulions réconcilier pratique et théorie afin de clarifier ou d'établir certaines justifications. Nous pensons y être parvenu sur plusieurs points, mais il subsiste encore des questions et des voies de recherche ouvertes. Elles expliquent aussi la préposition « vers » dans le titre de cette thèse.

Des transformations aux coûts. Pour réaliser un évaluateur partiel qui repose sur la formalisation que nous avons présentée, il nous manque une méthodologie et des outils qui nous aident à mettre en correspondance transformations et comparaisons de coût. Ces développements sont indispensables pour pouvoir traiter rigoureusement une application de taille réelle (pas simplement BOOL ou ZML!). Des résultats préliminaires en programmation logique nous laissent penser que notre approche est viable[1] et que cette thèse n'était pas un vain exercice de style[2].

Transformations élémentaires globales. Nous n'avons envisagé que des transformations élémentaires *locales*, qui agissent sur une zone ponctuelle du programme (pliage, dépliage, réécriture suivant les lois algébriques). Il faudrait aussi prendre en compte les transformations élémentaires *globales*, qui nécessitent un parcours itératif, comme par exemple pour le calcul d'un point fixe.

Correction des transformations de pliage/dépliage. Il reste encore à affaiblir davantage la condition de linéarité qui permet de ramener une transformation de pliage/dépliage quelconque en une transformation de dépliage/pliage faible (cf. §5.4.2, et note de bas de page numéro 1). Une autre possibilité est de reprendre certains des outils proposés et mis en œuvre pour le dépliage/pliage faible [Kot80], et de les appliquer à des transformations de pliage/dépliage plus générales.

Affaiblissement du critère de Higman-Kruskal. On peut chercher à réduire le coût d'évaluation du critère d'arrêt de Higman-Kriskal en affaiblissant la condition de plongement. Il faut s'appuyer sur le fait que la suite des états d'un programme transformé n'est pas quelconque (cf. §6.2.4, coût d'évaluation).

Emploi des informations de typage. La connaissance des types permet de meilleures compilations ; c'est le cas également de l'évaluation partielle. Par exemple, l'évaluation partielle généralisée peut déduire $x = 3$ des contraintes $2 < x$ et $x < 4$ à condition de savoir que x est entier. Pour inscrire la notion de typage dans notre formalisation, il faut faire en sorte qu'elle soit la plus transparente possible pour la modélisation de l'exécution et de la performance, mais qu'elle puisse néanmoins être prise en compte dans les transformations.

---

[1] Nous pouvons exprimer à l'aide de schémas de programmes réguliers plusieurs sémantiques de la programmation logique (sémantique procédurale [JM84, PP91a], déclarative [AvE82, Llo87], multi-ensemble [KK88a, KK88b]) en faisant simplement varier une ou deux règles de lois algébriques (commutativité et idempotence de la conjonction sur des séquences, des ensembles ou des multi-ensembles de substitutions). Ces schémas peuvent aussi bien modéliser l'exécution de la SLD-résolution PROLOG, qu'un parcours en largeur d'abord ou un approfondissement iteratif en profondeur d'abord (depth first iterative deepening) [ST85, Kor85b, Kor85a]. Ils permettent également de représenter la performance d'une compilation (simple) dans la WAM [War83, AK90] ou de celle d'une exécution par un interprète abstrait. Établir qu'une transformation fait décroître un coût nécessite toutefois d'assez longues preuves par récurrence sur la structure interne des termes manipulés par le programme, dont l'implantation en mémoire, dans le cas de la WAM, dépend a priori du passé de l'exécution.

Par ailleurs, la formulation en termes de schémas de programmes met en évidence et autorise de nouvelles transformations, tant théoriques que pratiques. Elle permet par exemple le pliage d'une disjonction, qui était masqué auparavant par les spécificités du langage. Elle permet aussi naturellement le pliage d'une conjonction ; bien qu'abondamment traité [LS91, GS91, PP91a, . . .] depuis la mise en œuvre l'évaluation partielle en programmation logique [Kom82, TS84], ce pliage est boudé par les évaluateurs partiels PROLOG, qui le limitent en pratique au pliage d'un simple atome.

D'autre part, la structure des termes se prête naturellement au critère de Higman-Kruskal. Pour des résultats automatiques intéressants, il faut néanmoins l'étendre — il faut faire jouer non seulement l'associativité et la commutativité, mais aussi la distributibité (cf. §6.2.4, coût d'évaluation). Il faut aussi étendre la fonction de généralisation de l'algorithme de redémarrage généralisé, afin de traiter les conjonctions et les disjonctions comme des listes associatives (éventuellement commutatives ou idempotentes). Enfin, pour pouvoir dérouler les boucles au maximum, il faut faire une transformation préliminaire (en pratique, virtuelle) qui accroît le cardinal de la signature algébrique, tout en conservant sa finitude.

Comme preuve de l'intérêt de notre approche, nous espérons, à l'aide de ces outils, pouvoir par exemple transformer *naturellement* et *automatiquement* la spécification déclarative d'un tri (énumération de permutations et vérification qu'elle est triée), de complexité exponentielle, en un tri par insertion, de complexité carrée.

[2] Ces résultats préliminaires (car encore partiels) ont en fait précédé la formalisation qui fait l'objet de cette thèse.

Ces perspectives ne doivent pas masquer le fait que l'évaluation partielle est soumise à des limitations, qu'il faut savoir assumer avant de se lancer dans cette entreprise.

Limites à l'optimisation par évaluation partielle.

Pour conclure, nous commentons des limites importantes de l'évaluation partielle qui remettent en cause, au moins sur le plan théorique, son bien-fondé. Nous suggérons néanmoins un compromis de « réhabilitation ».

Limites pratiques.

Les limites pratiques ne sont pas propres à l'évaluation partielle. Elles sont générales à tout système qui veut manipuler des programmes ou s'exprimer sur leur performance.

- **Taille du problème.** Avant d'affirmer qu'une évaluation partielle réalise effectivement une optimisation, il faut déjà modéliser fidèlement l'exécution et la performance d'une implémentation donnée, garantir que les transformations améliorent un coût et démontrer que l'implémentation des transformations est correcte. Or, ce sont des choses que l'on ne sait pas encore très bien faire à grande échelle. On peut imaginer ce que cela représente en considérant tout ce qu'il a fallu préciser pour des langages aussi triviaux que BOOL ou ZML.

- **Modélisation imparfaite.** La compléxité et la nature des processeurs et systèmes d'exploitation font l'on ne connaît qu'une approximation du « phénomène » d'exécution. Si l'on ajoute à cela les compromis indispensables en pratique pour définir un modèle d'exécution exploitable, on ne peut en toute rigueur affirmer qu'un programme est meilleur qu'un autre.

- **Effort inutile.** On peut aussi remarquer qu'il nous a fallu un effort non-négligeable pour obtenir un évaluateur partiel optimal de ZML. En fait, la compilation optimale en termes de graphe était relativement simple ; le retour à ZML fut plus compliqué. Or, ce retour est inutile si le programme doit ensuite être compilé, car il vaut mieux directement exploiter la structure intermédiaire de graphe. Autrement dit, l'optimisation par évaluation partielle peut devoir faire un travail en pratique inutile, simplement parce qu'il lui faut rester dans le même langage. Cette entrave peut se muer en limite pratique lorsque croît la complexité du problème.

Limites factuelles.

Par limite factuelle, nous entendons un fait notable, une action qu'il n'est pas possible ou qu'il n'est pas souhaitable de réaliser.

- **Connaissance de l'évaluateur partiel.** L'ambition des hommes et la maîtrise croissante des méthodes de programmation font que les programmes sont de plus en plus volumineux, complexes et critiques (sensibles). Il est donc primordial d'automatiser leurs traitements. En conséquence, il ne faut pas avoir à connaître le fonctionnement d'un évaluateur partiel — lui même de plus en plus volumineux et complexe — et devoir écrire des programmes « adaptés ». En particulier, le programmeur ne doit pas se contraindre à employer des heuristiques.

- **Construction à partir d'une spécification sémantique.** Il n'est pas suffisant de connaître la sémantique d'un langage pour réaliser un évaluateur partiel satisfaisant, car on ne dispose pas d'informations sur l'exécution. Il n'est donc pas envisageable de générer automatiquement un évaluteur partiel à partir de la seule spécification abstraite d'un langage.

- **Portabilité.** Un des arguments en faveur de l'évaluation partielle est la portabilité : une fois le programme optimisé par évaluation partielle, on peut l'exploiter sur plusieurs plateformes qui implémentent le langage. Pourtant, cet argument n'est pas recevable car la performance dépend de la machine d'exécution ; il n'existe pas *une* mais *des* évaluations partielles, suivant le modèle de performance considéré. Parler d'évaluation partielle « dans l'absolu » n'a pas de sens.

  Par exemple, les implémentations de PROLOG basé sur la WAM sont efficaces avec des prédicats allant typiquement jusqu'à quatre ou cinq arguments. Au delà, la gestion des arguments devient coûteuse et il est plus rentable de construire des données structurées, que l'on passe en paramètres (ils sont alors moins nombreux) et que l'on « détruit » dans le prédicat appelé. Dans le cas contraire, on peut avoir une perte de performance par rapport au programme d'origine [Sah91a, §2.4.3, p. 31]. C'est un problème qu'il ne faut pas négliger car on constate souvent, notamment en PROLOG, un accroissement d'arité lors des transformations d'évaluation partielle.

  Du reste, la question se traduit également en termes d'efforts de programmation pour réaliser un évaluateur partiel. Par exemple, il peut être fondamental ou inutile de faire apparaître des récursions terminales, suivant que la machine d'exécution en tient compte ou non.

Limites intrinsèques.

Parmi les limites intrinsèques de l'évaluation partielle, certaines sont spécifiques mais d'autres sont générales à toutes les techniques d'optimisation.

- **Indécidabilité.** La terminaison et l'équivalence des programmes sont des problèmes indécidables. Dans la mesure où une évaluation partielle q est avant tout un programme équivalent au programme original p, on ne peut généralement pas connaître *toutes* les solutions $q \in Evap(p)$ d'un problème d'évaluation partielle. Cela vaut tout autant pour les évaluations partielles optimales et minimale.

- **Pas d'optimum ou pas d'optimum atteignable.** Comme nous l'avons vu à la section §4.5, il n'existe pas en général d'évaluation partielle optimale, forte ou faible. C'est le cas également de l'évaluation partielle minimale, excepté pour un programme de domaine fini. Nous avons malgré tout donné des méthodologies pour établir qu'une évaluation partielle est optimale ou minimale, et indiqué comment transporter des résultats d'optimalité d'un langage dans un autre — le langage de départ étant, en pratique, choisi (ou créé) pour sa simplicité. Cependant, cela reste des méthodes « manuelles », spécifiques. Les procédés plus ou moins automatiques que nous avons recensés échouent sur ce problème.

- **Pouvoir d'optimisation.** L'évaluation partielle q est recherchée parmi les programmes du même langage que p. Même si l'on atteint un optimum, on ne garantit pas qu'une compilation vers un autre langage, notamment plus proche de la machine d'exécution réelle, n'aurait pas permis d'atteindre des performances encore meilleures. C'est naturel car « optimiser puis compiler » est *en théorie* moins bon que « compiler puis optimiser ». Rester dans le même langage est une contrainte qui borne la palette d'améliorations.

  Outre l'exemple formel que nous avons donné à la section §4.7, on imagine aisément qu'un programme dans un langage purement fonctionnel peut avoir une compilation plus efficace qui utilise des effets de bord, optimisation inaccessible à l'évaluation partielle. On peut tenir un discours identique, suivant que l'on dispose ou non de tableaux.

Extrême et conciliation.

Ces arguments, poussés à l'extrême, suggèrent que l'on traduise systématiquement un programme en code compilé et que l'on effectue ensuite sur ce code les optimisations. C'est en quelque sorte ce que réalisent déjà les passes d'optimisation en aval d'un compilateur. C'est aussi revenir aux sources de l'évaluation partielle, dont les idées originelles sont nées dans des compilateurs [Ers88]. Néanmoins, une traduction brute perd des informations. En particulier, tous les renseignements d'ordre sémantique qu'exploite l'évaluation partielle généralisée disparaissent dans le binaire. Ce point avait d'ailleurs été indirectement soulevé en 1987 parmi les nombreux « challenging problems in partial evaluation and mixed computation » [Jon88], mais il semble être resté dans l'ombre.

Il faut en fait trouver une voie médiane de « *compilation partielle* », qui préserve les informations de haut niveau manipulées d'ordinaire par les évaluateurs partiels, mais qui n'interdit pas les optimisations de bas niveau des compilateurs. D'ailleurs, si un évaluateur partiel se donne la peine de modéliser et d'optimiser selon une machine d'exécution qui est un compilateur, il ne doit pas avoir beaucoup plus de mal à générer directement le code optimisé.

Même cette solution reste extrême quand on connait le succès que remporte malgré tout l'évaluation partielle. En pratique, l'évaluation partielle produit très souvent des programmes (bien) meilleurs que les programmes originaux, et cela au sens de n'importe quelle implémentation courante. Les cas de perte de performance sont rares. Il y a donc une réalité immanente dans l'optimisation par évaluation partielle.

C'est pourquoi nous suggérons que l'on formalise une évaluation partielle en fonction de *classes* de modèles de performance. Une telle classe peut par exemple être compatible avec le remplacement d'une expression statique par sa valeur ou avec le dépliage. Ensuite, pour chaque plateforme sur laquelle on désire employer le résultat d'une évaluation partielle, il ne reste qu'à s'assurer qu'elle respecte bien la classe de modèles de performance correspondante. C'est dans cette direction que nous voyons un avenir plus sûr pour l'évaluation partielle.

Annexe D

Démonstrations Auxiliaires

Pour éviter d'encombrer inutilement le texte, nous avons regroupé dans cette annexe la démonstration de la plupart des propositions élémentaires énoncées dans ce document ; fastidieuses, elles n'apportent pas de compréhension supplémentaire des phénomènes. Nous avons aussi placé ici certaines démonstrations « techniques ». La numérotation des sections suit celle du document principal, que l'on préfixe simplement de la lettre D.

D.2 Sémantique et exécution.

Démonstration (définition 2.4 page 38). $Dom(M_s) = Dom(M_c \circ (T_{\mathcal{P}}, T_{\mathcal{D}})) = (T_{\mathcal{P}}^{-1}, T_{\mathcal{D}}^{-1})(Dom(M_c))$ et $Dom(M_c) = Dom(L_c)$ et $Dom(L_s) = Dom(T_{\mathcal{R}} \circ L_c \circ (T_{\mathcal{P}}, T_{\mathcal{D}})) = (T_{\mathcal{P}}^{-1}, T_{\mathcal{D}}^{-1})(Dom(T_{\mathcal{R}} \circ L_c))$ et $Dom(T_{\mathcal{R}} \circ L_c) = Dom(L_c)$ car $T_{\mathcal{R}}$ est une application, donc $Dom(M_s) = Dom(L_s)$. $\square$

Démonstration (définition 2.9 page 67). Les opérations de produit $D_1 \times \cdots \times D_n$ et de construction fonctionnelle $D_1 \rightarrow D_2$, avec les ordres produits et fonctionnels correspondants, préservent le caractère d'ensemble complet partiellement ordonné [Mos90]. Les domaines de $\boldsymbol{D}'$ sont donc bien des cpos. De même, les opérations (abstraction, application, produit, projection, test, etc.) préservent la continuité [Mos90]. Par conséquent, $\boldsymbol{D}'$ est bien une F'-algèbre continue. $\square$

D.3 Mesure de performance.

Démonstration (définition 3.7 page 78). $q \ (\bigotimes_{i \in I} \preccurlyeq_i)_{(\mu_i)_{i \in I}} \ q'$ ssi $(\mu_i(q))_{i \in I} \ (\bigotimes_{i \in I} \preccurlyeq_i) \ (\mu_i(q'))_{i \in I}$ ssi $\forall i \in I \ \ \mu_i(q) \preccurlyeq \mu_i(q')$ ssi $\forall i \in I \ \ q \preccurlyeq_{i,\mu_i} q'$ ssi $q \bigcap_{i \in I} \preccurlyeq_{i,\mu_i} q'$. $\square$

Démonstration (définition 3.8 page 79). $q \ (\bigotimes_{i \in I}^{\text{lex}} \preccurlyeq_i)_{(\mu_i)_{i \in I}} \ q'$ ssi $(\mu_i(q))_{i \in I} \ (\bigcup_{i \in I}((\bigotimes_{j \in I, j < i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j \in I, j > i} \approx_{j,\text{triv}})) \cup (\bigotimes_{i \in I} \preccurlyeq_i)) \ (\mu_i(q'))_{i \in I}$ ssi $q \ (\bigcup_{i \in I}((\bigcap_{j \in I, j < i} \preccurlyeq_{j,\mu_j}) \cap \prec_i \cap (\bigcap_{j \in I, j > i} \approx_{j,\text{triv}})) \cup (\bigcap_{i \in I} \preccurlyeq_{i,\mu_i})) \ q'$ $\square$

Démonstration (définition 3.9 page 80). $Dom(\mu) = Dom(\chi \circ M) = M^{-1}(Dom(\chi)) = M^{-1}(\mathcal{T}) = Dom(M)$ car χ est totale. Or $Dom(M) = Dom(L)$ par définition. □

Démonstration (définition 3.11 page 87). Les ensembles $(D_{s,\mathcal{C}}, \leqslant_{s,\mathcal{C}}, \perp_{s,\mathcal{C}})$ sont les cpos produits de $(D_s, \leqslant_s, \perp_s)$ et $(C, \leqslant_C, \perp_C)$. Puisque f_D et χ_f sont continues, et la formation d'une paire préserve la continuité, f_{D_c} est continue. □

Démonstration (définition 3.12 page 87). L'addition est une opération continue (au sens de l'ordre) dans le cpo $(C, \leqslant_C, \perp_C)$. Le produit, fonction de $\mathbb{N}_\perp \times C \to C$, est aussi continu. Puisque c_f et $N_{f,1}, \ldots, N_{f,n}$ sont continues, χ_f ainsi défini est également continu et le résultat précédent s'applique. □

Démonstration (proposition 3.1 page 98). $f \preccurlyeq_{\text{moy-pfini-lim}} g$ ssi $\forall \varepsilon > 0\ \exists h \in \mathcal{C}^D_{\text{norm}}\ \|h\| \leqslant \varepsilon$ et $\exists \Delta$ fini $\subset \mathcal{D}\ \forall \Delta'$ fini $\supset \mathcal{D}\ \sum_{d\in\Delta'} f(d) \leqslant \sum_{d\in\Delta'} g(d) + \sum_{d\in\Delta'} h(d)$. Avec $c = \sum_{d\in D} h(d)$, ou $h(d_0) = c$, $h_{|D\setminus\{d_0\}} = 0$ pour un $d_0 \in D$, on a $f \preccurlyeq_{\text{moy-pfini-lim}} g$ ssi $\forall \varepsilon > 0\ \exists c \in \mathcal{C}\ \|c\| \leqslant \varepsilon$ et $\exists \Delta$ fini $\subset D\ \forall \Delta'$ fini $\supset \Delta$ $\sum_{d\in\Delta'} f(d) \preccurlyeq \sum_{d\in\Delta'} g(d) + c$.

Soit $c_0 = \sum_{d\in D} h(d) - f(d) \succcurlyeq 0$ alors $\forall \varepsilon > 0\ \exists \Delta$ fini $\subset D$ $\|c_0 - \sum_{d\in\Delta}(h-f)(d)\| \leqslant \varepsilon$ et $\|\sum_{d\in D\setminus\Delta} |h(d) - f(d)|\,\| \leqslant \varepsilon$. Soit $c = c_0 + \sum_{d\in\Delta}(f-h)(d) + \sum_{d\in D\setminus\Delta} |f(d) - h(d)|$, alors $\forall \Delta'$ fini $\supset \Delta$ $\sum_{d\in\Delta'} h(d) \preccurlyeq \sum_{d\in\Delta'} g(d)$, donc $\sum_{d\in\Delta'} f(d) \preccurlyeq \sum_{d\in\Delta'} g(d) + \sum_{d\in\Delta'} f(d) - h(d) \preccurlyeq \sum_{d\in\Delta'} g(d) + \sum_{d\in\Delta}(f-h)(d) + \sum_{d\in D\setminus\Delta} |f(d) - h(d)| = \sum_{d\in\Delta'} g(d) - c_0 + c \preccurlyeq \sum_{d\in\Delta'} g(d) + c$ car $c_0 \succcurlyeq 0$, or $\|c\| \leqslant \|c_0 - \sum_{d\in\Delta}(h-f)(d)\| + \|\sum_{d\in D\setminus\Delta} |f(d) - h(d)|\| \leqslant 2\varepsilon$, donc $f \preccurlyeq_{\text{moy-lim}} g$. □

Démonstration (définition 3.35 page 104).

- Soient $f, g \in \mathcal{C}^D$. Alors $f \prec_{\text{abs}} g \Rightarrow f \not\succcurlyeq_{\text{stat}} g$ ssi $f \preccurlyeq_{\text{abs}} g \wedge g \not\preccurlyeq_{\text{abs}} f \Rightarrow g \not\preccurlyeq_{\text{stat}} f$ ssi $f \preccurlyeq_{\text{abs}} g \wedge g \preccurlyeq_{\text{stat}} f \Rightarrow g \preccurlyeq_{\text{abs}} f$.
- La contraposée de $f \preccurlyeq_{\text{abs}} g \wedge g \not\preccurlyeq_{\text{abs}} f \Rightarrow g \not\preccurlyeq_{\text{stat}} f$ s'écrit aussi $g \preccurlyeq_{\text{stat}} f \Rightarrow g \preccurlyeq_{\text{abs}} f \vee g \not\succcurlyeq_{\text{abs}} f$; en terme de graphes de relation, cela signifie $\preccurlyeq_{\text{stat}} \subset \preccurlyeq_{\text{abs}} \cup \not\succcurlyeq_{\text{abs}}$.
- $\prec_{\text{abs}} \subset \not\succcurlyeq_{\text{stat}}$ ssi $\not\preccurlyeq_{\text{abs}} \supset \succcurlyeq_{\text{stat}}$ ssi $\preccurlyeq_{\text{stat}} \subset \not\preccurlyeq_{\text{abs}}$.
- $\prec_{\text{abs}} = \preccurlyeq_{\text{abs}} \cap \not\succcurlyeq_{\text{abs}} \subset \preccurlyeq_{\text{stat}}$ par stabilité et $\prec_{\text{abs}} \subset \not\succcurlyeq_{\text{stat}}$ par discrimination donc $\prec_{\text{abs}} \subset \preccurlyeq_{\text{stat}} \cap \not\succcurlyeq_{\text{stat}} = \prec_{\text{stat}}$. □

Démonstration (section 3.5.6 page 107).

- Si $(\preccurlyeq_1 \cap \preccurlyeq_2)_{\text{stat}} \subset \preccurlyeq_{1,\text{stat}} \cap \preccurlyeq_{2,\text{stat}}$ et $\preccurlyeq_1 \subset \preccurlyeq_2$ alors $\preccurlyeq_{1,\text{stat}} = (\preccurlyeq_1 \cap \preccurlyeq_2)_{\text{stat}} = \preccurlyeq_{1,\text{stat}} \cap \preccurlyeq_{2,\text{stat}} \subset \preccurlyeq_{2,\text{stat}}$.
- Réciproquement si $\forall \preccurlyeq_1, \preccurlyeq_2$ $(\preccurlyeq_1 \subset \preccurlyeq_2 \Rightarrow \preccurlyeq_{1,\text{stat}} \subset \preccurlyeq_{2,\text{stat}})$, alors $\preccurlyeq_1 \cap \preccurlyeq_2 \subset \preccurlyeq_1$ donc $(\preccurlyeq_1 \cap \preccurlyeq_2)_{\text{stat}} \subset \preccurlyeq_{1,\text{stat}}$. De même , $\preccurlyeq_1 \cap \preccurlyeq_2 \subset \preccurlyeq_2$ donc $(\preccurlyeq_1 \cap \preccurlyeq_2)_{\text{stat}} \subset \preccurlyeq_{2,\text{stat}}$. Par conséquent $(\preccurlyeq_1 \cap \preccurlyeq_2)_{\text{stat}} \subset \preccurlyeq_{1,\text{stat}} \cap \preccurlyeq_{2,\text{stat}}$. □

Démonstration (proposition 3.2 page 108).

(1) Si $c \preccurlyeq c'$ et $c' \preccurlyeq c$ alors $0 \preccurlyeq c' - c \preccurlyeq 0$ par additivité, donc $\|c' - c\| \leqslant 0$ par monotonie, donc $c' - c = 0$: la relation $\preccurlyeq$ est antisymétrique. □

(2) Si $\nabla \subset \preccurlyeq_{\text{abs}} \subset \preccurlyeq_{\text{stat}}$ car $\preccurlyeq_{\text{abs}}$ est réflexif lorsque $\preccurlyeq$ l'est (tabl. 3.3).

(3) Si $f \preccurlyeq_{\text{abs}} g \preccurlyeq_{\text{stat}} f$ alors $f \preccurlyeq_{\text{stat}} g \preccurlyeq_{\text{stat}} f$ par stabilité, donc $f = g$ par antisymétrie donc $f \approx_{\text{abs}} g$ par réflexivité.

(4) C'est évident sur les définitions de la fusion forte et faible.

(5) Soit $(D_i)_{i \in I}$ une famille d'ensembles de données. $\forall J \subset I$ on pose $\Delta_J = \bigcap_{j \in J} D_j \setminus \bigcup_{i \in I \setminus J} D_i$, alors $(\Delta_J)_{J \subset I}$ est une partition de $\bigcup_{i \in I} D_i$.

- $\forall i \in I$ on pose $J_i = \{J \subset I \mid i \in J\}$ et l'on a $\bigcup_{J \subset J_i} \Delta_J = D_i$. En effet $\forall J \subset J_i\ \Delta_J \subset D_i$. D'autre part on pose $J_x = \{j \in I \mid x \in D_j\}$. Alors $\forall x \in D_i\ J_x \in J_i$ et $x \in \Delta_{J_x}$ donc $x \in \bigcup_{J \subset J_i} \Delta_J$. Par conséquent $\bigcup_{i \in I} D_i = \bigcup_{i \in I} \bigcup_{J \subset J_i} \Delta_J = \bigcup_{J \subset I} \Delta_J$.
- Supposons $J \neq J'$, c.-à-d. $\exists j \in J\ j \notin J'$ (l'autre cas est symétrique). Si $x \in \Delta_J$ alors $x \in D_j$ car $j \in J$ et $j \notin I \setminus J$, et si $x \in \Delta_{J'}$ alors $x \notin D_j$ car $j \in J' \setminus I$, donc $\Delta_J \cap \Delta_{J'} = \varnothing$.

Par conséquent $(\Delta_J)_{J \subset I}$ est une partition de $\bigcup_{i \in I} D_i$. On a $\cap_{i \in I} \preccurlyeq_{|D_i} = \cap_{i \in I} \preccurlyeq_{|\cup_{J \subset J_i} \Delta_J} \subset \cap_{i \in I} \cap_{J \subset J_i} \preccurlyeq_{|\Delta_J}$ par compatibilité avec la restriction, $= \cap_{J \subset I} \preccurlyeq_{|\Delta_J} = \preccurlyeq_{|\cup_{J \subset I} \Delta_J}$ par compatibilité avec la fusion faible, $= \preccurlyeq_{|\cup_{i \in I} D_i}$. Il y a donc compatibilité avec la fusion forte.

(6) Considérons tout d'abord la composabilité à droite.

[$\Rightarrow$] On voit sur les définitions que si $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est fortement composable à droite, alors il l'est aussi faiblement. D'autre part, si $f \preccurlyeq_D f'$ alors $\forall D' \subset D$ on considère $g = Id_{|D'}$ la fonction identité restreinte à D' et $h = 0_{|D'}$ la fonction constante nulle sur D'. Puisque $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est fortement composable à droite, on a $h + f \circ g \preccurlyeq_{g^{-1}(D)} h + f \circ g$, donc $0 + f \circ Id_{|D'} \preccurlyeq_{Id^{-1}_{|D'}(D)} 0 + f' \circ Id_{|D'}$, donc $f_{|D'} \preccurlyeq_{D'} f'_{|D'}$ car $Id^{-1}_{|D'}(D) = D'$, c.-à-d. $f \preccurlyeq_{|D'} f'$. Par conséquent $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est compatible avec la restriction du support.

[$\Leftarrow$] Supposons $f \preccurlyeq_D f'$ et $g \in \mathcal{D}^{\mathcal{D}}$. Soit $g' = g_{|g^{-1}(D)}$. On a $Im(g') = Im(g) \cap D$ et $g^{-1}(D) = g^{-1}(Im(g'))$. Alors $f \preccurlyeq_{|Im(g')} f'$ car $Im(g') \subset D$ et $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est compatible avec la resctriction. Donc $\forall h \in \mathcal{C}^{Dom(g)}$ $h + f \circ g \preccurlyeq_{g^{-1}(Im(g'))} h + f' \circ g$ car $Im(g) \supset Im(g')$ et $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est faiblement composable à droite. Or $g^{-1}(D) = g^{-1}(Im(g'))$, donc $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est fortement composable à droite.

Considérons maintenant la composabilité à gauche.

[$\Rightarrow$] La composabilité forte implique aussi composabilité faible car si $Dom(h) = g(D)$ alors $g^{-1}(g(D)) = D$ car $Dom(g) = D$. D'autre part, si $f \preccurlyeq_D f'$ alors $\forall D' \subset D$ on considère les fonctions $g = Id_{|D'}$ et $h = 0_{|\mathcal{D}}$. Alors $Dom(h) = \mathcal{D}$ et $g^{-1}(Dom(h)) = D'$. Puisque $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est fortement composable à gauche, on a $f + h \circ g \preccurlyeq_{|g^{-1}(Dom(h))} f' + h \circ g$, donc $f + 0_{|D'} \preccurlyeq_{|D'} f' + 0_{|D'}$ donc $f \preccurlyeq_{|D'} f'$. Par conséquent, $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est compatible avec la restriction du support.

[$\Leftarrow$] Supposons $f \preccurlyeq_D f'$ et $g \in \mathcal{D}^D$ et $h \in \mathcal{C}^{\mathcal{D}}$. $Alors f + h \circ g \preccurlyeq_D f' + h \circ g$ car $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est faiblement composable à gauche. Or $g^{-1}(Dom(h)) \subset D$ car $Dom(g) = D$, donc $f + h \circ g \preccurlyeq_D f' + h \circ g$ car $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est compatible avec la resctriction. $Parconsequent (\preccurlyeq_D)_{D \subset \mathcal{D}}$ est fortement composable à gauche.

(7) Supposons $f \preccurlyeq_D f'$. $\forall g \in \mathcal{D}^{\mathcal{D}}$ si $Im(g) \supset D$ on pose $\Delta = \{D' \subset g^{-1}(D) \mid g_{|D'}$ bijection $\in D^{D'}\}$. Alors $\forall D' \in \Delta$ $((f \circ g_{|D'})(d'))_{d' \in D'} = (f(g(d')))_{d' \in D'} = (f(d))_{d \in D}$ et de même pour f', donc $f \circ g_{|D'} \preccurlyeq_{D'} f' \circ g_{|D'}$, donc $h + f \circ g \preccurlyeq_{|D'} h + f' \circ g$ par additivité pour $h \in \mathcal{C}^{Dom(g)}$, donc $h + f \circ g \preccurlyeq_{|\cup_{D' \in \Delta} D'} h + f' \circ g$ par fusion forte. Or $\bigcup_{D' \in \Delta} D' = g^{-1}(D)$, donc $(\preccurlyeq_D)_{D \subset \mathcal{D}}$ est faiblement composable à droite.

(8) si $f \preccurlyeq_D f'$ on pose $g = Id_{|D}$. On a bien $Im(g) \supset D$. Donc $\forall h \in \mathcal{C}^{Dom(g)} = \mathcal{C}^D$ on a $h + f \preccurlyeq h + f'$.

(9) [$\Leftarrow$] Si $f \preccurlyeq_D f'$ alors $f + h \circ g \preccurlyeq_D f' + h \circ g$ car $\preccurlyeq_D$ est additif.
[$\Rightarrow$] Réciproquement si $f \preccurlyeq_D f'$ on pose $g = Id_{|\mathcal{D}}$, alors $\forall h \in \mathcal{C}^{g(D)} = \mathcal{C}^D$ on a $f + h \preccurlyeq f' + h$. $\square$

D.3.6 Propriétés des relations de coûts.

Nous rassemblons ici les démonstrations des propriétés des relations de coût données à la section §3.6. Nous n'indiquons pas toujours de référence explicite lorsqu'une propriété est déduite grâce aux identités de la table 3.1, par compatibilité avec une construction de coût, ou à l'aide des liens entre propriétés établis par la proposition 3.2.

D.3.6.2 Propriétés des combinaisons de coûts.

Démonstration (proposition 3.3 page 109).

- Si $c_1 \preccurlyeq c_2 \prec c_3$ alors $c_1 \preccurlyeq c_2 \preccurlyeq c_3$ et $c_2 \not\succcurlyeq c_3$. Supposons $c_1 \succcurlyeq c_3$, alors $c_3 \preccurlyeq c_1 \preccurlyeq c_2$ contredit $c_2 \not\succcurlyeq c_3$, donc $c_1 \prec c_3$. Autrement dit $\preccurlyeq \cdot \prec \,\subset\, \prec$. On a symétriquement $\prec \cdot \preccurlyeq \,\subset\, \prec$.
- Si $c_1 \preccurlyeq c_1'$ et $c_2 \preccurlyeq c_2'$ et $c_2 \not\succcurlyeq c_2'$ alors $c_1 + c_2 \preccurlyeq c_1' + c_2'$ par additivité. Si $c_1 + c_2 \succcurlyeq c_1' + c_2'$ alors $c_2 \succcurlyeq c_2'$ par additivité, contredit $c_2 \not\succcurlyeq c_2'$, donc $c_1 + c_2 \not\succcurlyeq c_1' + c_2'$, c.-à-d. $\preccurlyeq + \prec \,\subset\, \prec$.
- Si $f \prec_{\mathrm{abs}} g$, c.-à-d. $f \preccurlyeq_A BSg$ et $f \not\succ_{\mathrm{abs}} g$, alors $f \preccurlyeq_{\mathrm{stat}} g$ par stabilité. Si $f \succcurlyeq_{\mathrm{stat}} g$, alors $f \succcurlyeq_{\mathrm{abs}} g$ par discrimination contredit $f \not\succ_{\mathrm{abs}} g$, donc $f \not\succcurlyeq_{\mathrm{stat}} g$, c.-à-d. $f \prec_{\mathrm{stat}} g$. $\square$

Réciproque. Les propriétés ont la même forme pour $\preccurlyeq$ comme pour $\succcurlyeq$.

Complémentaire. [$\not\preccurlyeq = \complement(\preccurlyeq)$]

- Additivité : Supposons $c_1 \not\preccurlyeq c_2$; si $c_1 + c_3 \preccurlyeq c_2 + c_3$, alors $c_1 \preccurlyeq c_2$ par additivité, contredit $c_1 \not\preccurlyeq c_2$, donc $c_1 + c_3 \not\preccurlyeq c_2 + c_3$.
- Homothéticité : Supposons $c_1 \not\preccurlyeq c_2$; si $\alpha.c_1 \preccurlyeq \alpha.c_2$ avec $\alpha > 0$ alors $(1/\alpha).\alpha.c_1 \preccurlyeq (1/\alpha).\alpha.c_2$ par homothéticité, contredit $c_1 \not\preccurlyeq c_2$, donc $\alpha.c_1 \not\preccurlyeq \alpha.c_2$.
- Fusion forte (resp. faible) si compatibilité avec la restriction : Supposons $\forall i \in I \quad f \not\preccurlyeq_{|D_i} g$. Si $f \preccurlyeq_{|\cup_{i\in I} D_i} g$ alors $\forall i \in I \quad f \preccurlyeq_{|D_i} g$ par restriction contredit $f \not\preccurlyeq_{|D_i} g$ donc $f \not\preccurlyeq_{|\cup_{i\in I} D_i} g$.

Relation stricte. [$\prec = \preccurlyeq \cap \not\succcurlyeq$]

- Antisymétrie (triviale) : $\prec \cap \succ = \varnothing \subset \nabla$.
- Transitivité : $\prec \,\subset\, \preccurlyeq$ donc $\prec \cdot \prec \,\subset\, \prec \cdot \preccurlyeq$. Or $\prec \cdot \preccurlyeq \,\subset\, \prec$ (prop. 3.3).
- Additivité et homothéticité : elles sont préservées par réciproque, complémentaire et intersection.
- Discrimination : si $\preccurlyeq \,\subset\, \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$ alors $\prec = \preccurlyeq \cap \not\succcurlyeq \,\subset\, \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$.

Inclusion.

- Réflexivité : $\nabla \subset \preccurlyeq_1 \subset \preccurlyeq_2$.
- Antisymétrie : $\preccurlyeq_1 \cap \succcurlyeq_1 \subset \preccurlyeq_2 \cap \succcurlyeq_2 \subset \nabla$.
- Totalité : $\mathcal{C}^2 = \preccurlyeq_1 \cup \succcurlyeq_1 \subset \preccurlyeq_2 \cup \succcurlyeq_2$.
- Stabilité : $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_1 \subset \preccurlyeq_2$.
- Discrimination : $\preccurlyeq_1 \subset \preccurlyeq_2 \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$.

Intersection.

- Réflexivité : si $\forall i \in I \quad \nabla \subset \preccurlyeq_i$ alors $\nabla \subset \bigcap_{i\in I} \preccurlyeq_i$.
- Antisymétrie : si $\exists j \in I \quad \preccurlyeq_j \cap \succcurlyeq_j \subset \nabla$ alors $(\bigcap_{i\in I} \preccurlyeq_i) \cap (\bigcap_{i\in I} \succcurlyeq_i) \subset \preccurlyeq_j \cap \succcurlyeq_j \subset \nabla$.
- Transitivité : si $f\ (\bigcap_{i\in I} \preccurlyeq_i)\ g\ (\bigcap_{i\in I} \preccurlyeq_i)\ h$ alors $\forall i \in I \quad f \preccurlyeq_i g \preccurlyeq_i h$.
- Additivité : $(\bigcap_{i\in I} \preccurlyeq_i) + \nabla \subset \bigcap_{i\in I}(\preccurlyeq_i + \nabla) \subset \bigcap_{i\in I} \preccurlyeq_i$.
- Homothéticité : $\alpha.(\bigcap_{i\in I} \preccurlyeq_i) \subset \bigcap_{i\in I} \alpha.\preccurlyeq_i \subset \bigcap_{i\in I} \preccurlyeq_i$.
- Stabilité : si $\forall i \in I \quad \preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_i$ alors $\preccurlyeq_{\mathrm{abs}} \subset \bigcap_{i\in I} \preccurlyeq_i$.
- Discrimination : si $\exists j \in I \quad \preccurlyeq_j \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$ alors $\bigcap_{i\in I} \preccurlyeq_i \preccurlyeq_i \subset \preccurlyeq_j \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$.
- Restriction : $(\bigcap_{i\in I} \preccurlyeq_i)_\Delta = \bigcap_{i\in I} \preccurlyeq_{i\Delta} \subset \bigcap_{i\in I}(\bigcap_{\Delta'\subset\Delta} \preccurlyeq_{i|\Delta'}) = \bigcap_{\Delta'\subset\Delta} \bigcap_{i\in I} \preccurlyeq_{i|\Delta'} = \bigcap_{\Delta'\subset\Delta}(\bigcap_{i\in I} \preccurlyeq_i)_\Delta$.
- Fusion : $\bigcap_{j\in J}(\bigcap_{i\in I} \preccurlyeq_i)_{|D_j} = \bigcap_{i\in I}\bigcap_{j\in J} \preccurlyeq_{i|D_j} \subset \bigcap_{i\in I} \preccurlyeq_{i|\cup_{j\in J} D_j} = (\bigcap_{i\in I} \preccurlyeq_i)_{|\cup_{j\in J} D_j}$.
- Composabilité à droite : Si $\forall i \in I \quad f \preccurlyeq_{i,D} f'$ alors $\forall i \in I \quad h + f \circ g \preccurlyeq_{i,g^{-1}(D)} h + f' \circ g$.

Réunion.

- Réflexivité : si $\exists i \in I \quad \nabla \subset \preccurlyeq_i$ alors $\nabla \subset \bigcup_{i\in I} \preccurlyeq_i$.
- Transitité : si $\exists i \in I \quad f \preccurlyeq_i g$ et $\exists j \in J \quad g \leqslant_j h$ alors $\exists k \in I \quad f \preccurlyeq_k g \preccurlyeq_k h$ avec $\preccurlyeq_i \cup \preccurlyeq_j \subset \preccurlyeq_k$.
- Totalité : si $\exists j \in I \quad \preccurlyeq_j \cup \succcurlyeq_j = \mathcal{C}^2$ alors $(\bigcup_{i\in I} \preccurlyeq_i) \cup (\bigcup_{i\in I} \succcurlyeq_i) = \bigcup_{i\in I}(\preccurlyeq_i \cup \succcurlyeq_i) \supset \preccurlyeq_j \cup \succcurlyeq_j = \mathcal{C}^2$.
- Additivité : $(\bigcup_{i\in I} \preccurlyeq_i) + \nabla = \bigcup_{i\in I}(\preccurlyeq_i + \nabla) \subset \bigcup_{i\in I} \preccurlyeq_i$.
- Homothéticité : $\alpha.(\bigcup_{i\in I} \preccurlyeq_i) \subset \bigcup_{i\in I} \alpha.\preccurlyeq_i \subset \bigcup_{i\in I} \preccurlyeq_i$.
- Stabilité : si $\exists i \in I \quad \preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_i$ alors $\preccurlyeq_{\mathrm{abs}} \subset \bigcup_{i\in I} \preccurlyeq_i$.
- Discrimination : si $\forall i \in I \quad \preccurlyeq_i \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$ alors $\bigcup_{i\in I} \preccurlyeq_i \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$.
- Restriction : $(\bigcup_{i\in I} \preccurlyeq_i)_\Delta = \bigcup_{i\in I} \preccurlyeq_{i\Delta} \subset \bigcup_{i\in I}\bigcap_{\Delta'\subset\Delta} \preccurlyeq_{i|\Delta'} \subset \bigcap_{\Delta'\subset\Delta}\bigcup_{i\in I} \preccurlyeq_{i|\Delta'} = \bigcap_{\Delta'\subset\Delta}(\bigcup_{i\in I} \preccurlyeq_i)_{|\Delta'}$.
- Composabilité à droite : Si $\exists i \in I \quad f \preccurlyeq_{i,D} f'$ alors $h + f \circ g \preccurlyeq_{i,g^{-1}(D)} h + f' \circ g$.

Produit.

- Réflexivité : $\nabla_{\Pi_{i\in I}\mathcal{C}_i} = \bigotimes_{i\in I} \nabla_{\mathcal{C}_i} \subset \bigotimes_{i\in I} \preccurlyeq_i$.
- Antisymétrie : $(\bigotimes_{i\in I} \preccurlyeq_i) \cap (\bigotimes_{i\in I} \succcurlyeq_i) = \bigotimes_{i\in I}(\preccurlyeq_i \cap \succcurlyeq_i) \subset \bigotimes_{i\in I} \nabla_{\mathcal{C}_i} = \nabla_{\Pi_{i\in I}\mathcal{C}_i}$.
- Transitivité : $(\bigotimes_{i\in I} \preccurlyeq_i) \cdot (\bigotimes_{i\in I} \preccurlyeq_i) = \bigotimes_{i\in I}(\preccurlyeq_i \cdot \preccurlyeq_i) \subset \bigotimes_{i\in I} \preccurlyeq_i$.
- Additivité : $(\bigotimes_{i\in I} \preccurlyeq_i) + (\bigotimes_{i\in I} \nabla_i) = \bigotimes_{i\in I}(\preccurlyeq_i + \nabla_i) \subset \bigotimes_{i\in I} \preccurlyeq_i$.
- Homothéticité : $\alpha.(\bigotimes_{i\in I} \preccurlyeq_i) = \bigotimes_{i\in I} \alpha.\preccurlyeq_i \subset \bigotimes_{i\in I} \preccurlyeq_i$.
- Stabilité : si $\forall i \in I \quad \preccurlyeq_{i,\mathrm{abs},D} \subset \preccurlyeq_{i,D}$ alors $(\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{abs},D} = \bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{abs},D} \subset \bigotimes_{i\in I} \preccurlyeq_{i,D}$ (cf. §3.6.3).
- Discrimination : $(f_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{abs},D}\ (g_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_{i,D})\ (f_i)_{i\in I}$ ssi $\forall i \in I \quad f_i \preccurlyeq_{i,\mathrm{abs},D} g_i \preccurlyeq_{i,D} f_i$ donc $\forall i \in I \quad g_i \preccurlyeq_{i,\mathrm{abs},D} f_i$ donc $(g_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{abs},D})\ (f_i)_{i\in I}$.
- Restriction : $(\bigotimes_{i\in I} \preccurlyeq_i)_\Delta = \bigotimes_{i\in I} \preccurlyeq_{i\Delta} \subset \bigotimes_{i\in I}(\bigcap_{\Delta'\subset\Delta} \preccurlyeq_{i|\Delta'}) = \bigcap_{\Delta'\subset\Delta}\bigotimes_{i\in I} \preccurlyeq_{i|\Delta'} = \bigcap_{\Delta'\subset\Delta} (\bigotimes_{i\in I} \preccurlyeq_i)_{|\Delta'}$.
- Fusion : $\bigcap_{j\in J}(\bigotimes_{i\in I} \preccurlyeq_i)_{|D_j} = \bigotimes_{i\in I}(\bigcap_{j\in J} \preccurlyeq_{i|D_j}) \subset \bigotimes_{i\in I} \preccurlyeq_{i|\cup_{j\in J} D_j} = (\bigotimes_{i\in I} \preccurlyeq_i)_{|\cup_{j\in J} D_j}$.
- Composabilité à droite : Si $\forall i \in I \quad f_i \preccurlyeq_{i,D} f_i'$ alors $h_i + f_i \circ g_i \preccurlyeq_{i,g_1^{-1}(D)} h_i + f_i' \circ g_i$.

Produit lexicographique. $[\ \preccurlyeq\ = \bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_i = \bigcup_{i\in I}((\bigotimes_{j\in I, j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I, j>i} \approx_{j,\mathrm{triv}})) \cup (\bigotimes_{i\in I} \preccurlyeq_i)\ ]$

- Réflexivité : $\nabla_{\Pi_{i\in I}\mathcal{C}_i} \subset \bigotimes_{i\in I} \preccurlyeq_i \subset \bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_i$.

- Antisymétrie : Soit $\preccurlyeq' = \bigcup_{i\in I}((\bigotimes_{j\in I,j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I,j>i} \approx_{j,\mathrm{triv}}))$. Alors $\bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_i \cap \bigotimes^{\mathrm{lex}}_{i\in I} \succcurlyeq_i = (\bigotimes_{i\in I} \preccurlyeq_i \cap \bigotimes_{i\in I} \succcurlyeq_i) \cup (\preccurlyeq' \cap \bigotimes_{i\in I} \preccurlyeq_i) \cup (\bigotimes_{i\in I} \preccurlyeq_i \cap \succcurlyeq') \cup (\preccurlyeq' \cap \succcurlyeq')$. Seul le premier de ces termes n'est pas nécessairement vide :
  - Le second terme est $(\bigcup_{i\in I}((\bigotimes_{j\in I,j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I,j>i} \approx_{j,\mathrm{triv}}))) \cap \bigotimes_{i\in I} \preccurlyeq_i = \bigcup_{i\in I}((\bigotimes_{j\in I,j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I,j>i} \approx_{j,\mathrm{triv}})) \cap (\bigotimes_{i\in I} \preccurlyeq_i) = \bigcup_{i\in I}(\bigotimes_{j\in I,j<i}(\preccurlyeq_j \cap \succcurlyeq_j) \otimes (\prec_i \cap \succcurlyeq_i) \otimes (\bigotimes_{j\in I,j>i} (\approx_{j,\mathrm{triv}} \cap \succcurlyeq_j))) = \bigcup_{i\in I} \varnothing = \varnothing$.
  - Le troisième terme est la relation réciproque du second, et donc vide.
  - Le quatrième est $\preccurlyeq' \cap \succcurlyeq' = \bigcup_{i\in I, i'\in I}((\bigotimes_{j\in I,j<i} \preccurlyeq_j) \otimes \prec_i \otimes (\bigotimes_{j\in I,j>i} \approx_{j,\mathrm{triv}})) \cap ((\bigotimes_{j'\in I,j'<i'} \succcurlyeq_{j'}) \otimes \succ_{i'} \otimes (\bigotimes_{j'\in I,j'>i'} \approx_{j',\mathrm{triv}}))$. Ces termes sont tous vides pour $i, i' \in I$: lorsque $i = i'$ le produit comporte $\prec_i \cap \succ_{i'} = \varnothing$; lorsque $i < i'$ le produit comporte $\prec_i \cap \succcurlyeq_{i'} = \varnothing$; lorsque $i > i'$ le produit comporte $\preccurlyeq_i \cap \succ_{i'} = \varnothing$.

  Par conséquent, $\bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_i \cap \bigotimes^{\mathrm{lex}}_{i\in I} \succcurlyeq_i = \bigotimes_{i\in I} \preccurlyeq_i \cap \bigotimes_{i\in I} \succcurlyeq_i \subset \nabla_{\Pi_{i\in I}\mathcal{C}_i}$ (voir ci-dessus).
- Transitivité : supposons $(c_i)_{i\in I} \preccurlyeq (c'_i)_{i\in I} \preccurlyeq (c''_i)_{i\in I}$. Il y a quatre cas :
  - $(\exists i\in I \quad \forall j<i \quad c_j \preccurlyeq_j c'_j$ et $c_i \prec_i c'_i)$ et $(\exists i'\in I \quad \forall j<i' \quad c'_j \preccurlyeq_j c''_j$ et $c'_{i'} \prec_{i'} c''_{i'})$. Alors $\exists i'' = \min(i,i')$ (par exemple $i'' = i'$) $\forall j < i'' \quad c_j \preccurlyeq_j c'_j \preccurlyeq c''_j$ et $c_{i''} \prec_{i''} c'_{i''} \preccurlyeq c''_{i''}$.
  - $(\exists i\in I \quad \forall j<i \quad c_j \preccurlyeq_j c'_j$ et $c_i \prec_i c'_i)$ et $(\forall i'\in I \quad c'_{i'} \preccurlyeq_{i'} c''_{i'})$, alors $\forall j<i \quad c_j \preccurlyeq_j c'_j \preccurlyeq_j c''_j$ et $c_i \prec_i c'_i \preccurlyeq_i c''_i$.
  - $(\forall i\in I \quad c_i \preccurlyeq_i c'_i)$ et $(\exists i'\in I \quad \forall j<i' \quad c'_j \preccurlyeq_j c''_j$ et $c'_{i'} \prec_{i'} c''_{i'})$. C'est le cas symétrique du cas précédent.
  - $(\forall i\in I \quad c_i \preccurlyeq_i c'_i)$ et $(\forall i'\in I \quad c'_{i'} \preccurlyeq_{i'} c''_{i'})$ alors $\forall i\in I \quad c_i \preccurlyeq_i c'_i \preccurlyeq_i c''_i$.

  Dans chaque cas, on remplit bien une des conditions du coût produit lexicographique : $(c_i)_{i\in I} \preccurlyeq (c''_i)_{i\in I}$.
- Additivité et homothéticité : elles sont préservées par réunion, intersection, relation stricte, et sont vérifiées par $\approx_{\mathrm{triv}}$.
- Stabilité : si $\forall i\in I \quad \preccurlyeq_{i,\mathrm{abs},D} \subset \preccurlyeq_{i,D}$ alors $(\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{abs},D} \subset \bigotimes_{i\in I} \preccurlyeq_{i,D} \subset \bigotimes^{\mathrm{lex}}_{i\in I} \preccurlyeq_{i,D}$.

D.3.6.3 Propriétés préservées par les coûts statiques déduits de coûts dynamiques.

Coût absolu. **[** $f \preccurlyeq_{\mathrm{abs}} g$ ssi $\forall d\in D \quad f(d) \preccurlyeq g(d)$ **]**

- « $(\nabla_\mathcal{C})_{\mathrm{abs},D} = \nabla_{\mathcal{C}^D}$ » : $f\ (\nabla_\mathcal{C})_{\mathrm{abs},D}\ g$ ssi $\forall d\in D \quad f(d) = g(d)$ ssi $f\ \nabla_{\mathcal{C}^D}\ g$.
- Antisymétrie : si $(\preccurlyeq \cap \succcurlyeq) \subset \nabla_\mathcal{C}$ alors $(\preccurlyeq \cap \succcurlyeq)_{\mathrm{abs},D} = \preccurlyeq_{\mathrm{abs},D} \cap \succcurlyeq_{\mathrm{abs},D} \subset (\nabla_\mathcal{C})_{\mathrm{abs},D} = \nabla_{\mathcal{C}^D}$.
- Transitivité : si $\forall d\in D \quad f(d) \preccurlyeq g(d) \preccurlyeq h(d)$ alors $\forall d\in D \quad f(d) \preccurlyeq h(d)$.
- Additivité : si $f \preccurlyeq_{\mathrm{abs}} g$ alors $\forall d\in D \quad f(d) \preccurlyeq g(d)$ donc $\forall d\in D \quad f(d) + h(d) \preccurlyeq g(d) + h(d)$.
- Homothéticité : si $\forall d\in D \quad f(d) \preccurlyeq g(d)$ alors $\forall d\in D \quad \alpha.f(d) \preccurlyeq \alpha.g(d)$ pour $\alpha > 0$.
- Stabilité : $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\mathrm{abs}}$.
- Discrimination : $\preccurlyeq_{\mathrm{abs}} \subset \preccurlyeq_{\mathrm{abs}} \cup \not\succcurlyeq_{\mathrm{abs}}$.
- Restriction : si $f \preccurlyeq_{\mathrm{abs},D|\Delta} g$ et $\Delta' \subset \Delta$ alors $f \preccurlyeq_{\mathrm{abs},D|\Delta'} g$.
- Fusion forte : $\forall i\in I \quad f \preccurlyeq_{\mathrm{abs},D|D_i} g$ ssi $\forall i\in I \quad \forall d\in D_i \quad f(d) \preccurlyeq g(d)$ ssi $\forall d\in \bigcup_{i\in I} D_i \quad f(d) \preccurlyeq g(d)$ ssi $f \preccurlyeq_{\mathrm{abs},D|\cup_{i\in I} D_i} g$.
- Composabilité : par la proposition 3.2 car $\preccurlyeq_{\mathrm{abs}}$ est additif et compatible avec la restriction.
- Complémentaire : si $D \neq \varnothing$ alors $f \not\preccurlyeq_{\mathrm{abs}} g$ ssi $\neg(\forall d\in D \quad f(d) \preccurlyeq g(d))$ ssi $\exists d\in D \quad f(d) \not\preccurlyeq g(d)$ si $\forall d\in D \quad f(d) \not\preccurlyeq g(d)$ ssi $f\ (\not\preccurlyeq)_{\mathrm{abs}}\ g$; si $D = \varnothing$ alors $\preccurlyeq_{\mathrm{abs},D} = \approx_{\mathrm{triv}} = (\not\preccurlyeq)_{\mathrm{abs},D}$.
- Intersection : $\forall d\in D \quad \forall i\in I \quad f(d) \preccurlyeq_i g(d)$ ssi $\forall i\in I \quad \forall d\in D \quad f(d) \preccurlyeq_i g(d)$.

- Réunion : $\forall d \in D \quad \exists i \in I \quad f(d) \preccurlyeq_i g(d)$ si $\exists i \in I \quad \forall d \in D \quad f(d) \preccurlyeq_i g(d)$.
- Produit : $(f_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{abs}}\ (g_i)_{i\in I}$ ssi $\forall d \in D \quad \forall i \in I \quad f_i(d) \preccurlyeq_i g_i(d)$ ssi $\forall i \in I \quad f_i \preccurlyeq_{i,\mathrm{abs}} g_i$ ssi $(f_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{abs}})\ (g_i)_{i\in I}$.

Coût moyen. [$f \preccurlyeq_{\mathrm{moy}} g$ ssi $\forall d \in D \quad \sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} g(d)$ avec D fini]

- Transitivité : si $\sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} g(d) \preccurlyeq \sum_{d\in D} h(d)$ alors $\sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} h(d)$.
- Totalité : si $\preccurlyeq$ est totale alors $\sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} g(d)$ ou $\sum_{d\in D} f(d) \succcurlyeq \sum_{d\in D} g(d)$.
- Additivité : si $f \preccurlyeq_{\mathrm{moy}} g$ alors $\sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} g(d)$ donc $\sum_{d\in D} f(d) + h(d) \preccurlyeq g(d) + h(d)$.
- Homothéticité : si $\sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} g(d)$ alors $\sum_{d\in D} \alpha.f(d) \preccurlyeq \sum_{d\in D} \alpha.g(d)$ pour $\alpha > 0$.
- Stabilité : si $\forall d \in D \quad f(d) \preccurlyeq g(d)$ alors $\sum_{d\in D} f(d) \preccurlyeq \sum_{d\in D} g(d)$ par additivité.
- Discrimination : si $f \preccurlyeq_{\mathrm{abs}} g \preccurlyeq_{\mathrm{moy}} f$ alors $\forall d_0 \in D \quad \sum_{d\in D} g(d) = \sum_{d\in D\setminus\{d_0\}} g(d) + g(d_0) \preccurlyeq \sum_{d\in D} f(d)$, or $\sum_{d\in D\setminus\{d_0\}} f(d) \preccurlyeq \sum_{d\in D\setminus\{d_0\}} g(d)$ par additivité, donc $g(d_0) \preccurlyeq f(d_0)$.
- Fusion faible : supposons $\forall i,j \in I \quad i \neq j \Rightarrow D_i \cap D_j = \varnothing$, alors I est fini de cardinal au plus $|D|$. On a $\forall i \in I \quad f \preccurlyeq_{\mathrm{moy},D|D_i} g$ ssi $\forall i \in I \quad \sum_{d\in D_i} f(d) \preccurlyeq \sum_{d\in D_i} g(d)$ alors $\sum_{i\in I}\sum_{d\in D_i} f(d) \preccurlyeq \sum_{i\in I}\sum_{d\in D_i} g(d)$ par additivité car I fini, ssi $\sum_{d\in(\cup_{i\in I} D_i)} f(d) \preccurlyeq \sum_{d\in(\cup_{i\in I} D_i)} g(d)$ car $\forall i,j \in I \quad i \neq j \Rightarrow D_i \cap D_j = \varnothing$, ssi $f \preccurlyeq_{\mathrm{moy},D|\cup_{i\in I} D_i} g$.
- Composabilité faible : par la proposition 3.2 car $\preccurlyeq_{\mathrm{moy}}$ est additif si $\preccurlyeq$ l'est.
- Réciproque, complémentaire, relation et équivalence : elles sont sur le modèle de $f\ (\succcurlyeq)_{\mathrm{moy}}\ g$ ssi $\sum_{d\in D} f(d) \succcurlyeq \sum_{d\in D} g(d)$ ssi $f \succcurlyeq_{\mathrm{moy}} g$.
- Intersection : $\sum_{d\in D} f(d)\ (\bigcap_{i\in I} \preccurlyeq_i)\ \sum_{d\in D} g(d)$ ssi $\forall i \in I \quad \sum_{d\in D} f(d) \preccurlyeq_i \sum_{d\in D} g(d)$.
- Produit : $(f_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{moy}}\ (g_i)_{i\in I}$ ssi $\forall i \in I \quad \sum_{d\in D} f_i(d) \preccurlyeq_i \sum_{d\in D} g_i(d)$ ssi $(f_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{moy}})\ (g_i)_{i\in I}$.

Coût maximum. [$f \preccurlyeq_{\max} g$ ssi $\exists d' \in D \quad \forall d \in D \quad f(d) \preccurlyeq g(d')$]

- Transitivité : si $\exists d_1 \in D \quad \forall d_2 \in D \quad f(d_2) \preccurlyeq g(d_1)$ et $\exists d_3 \in D \quad \forall d_4 \in D \quad g(d_4) \preccurlyeq h(d_3)$, alors $\forall d_2 \in D \quad f(d_2) \preccurlyeq g(d_1) \preccurlyeq h(d_3)$.
- Totalité : si $\preccurlyeq$ est totale et D est fini alors $f \preccurlyeq_{\max} g$ ssi $\max_{d\in D} f(d) \preccurlyeq \max_{d\in D} g(d)$.
- Homothéticité : si $\exists d' \in D \quad \forall d \in D \quad f(d) \preccurlyeq g(d')$ alors $\forall d \in D \quad \alpha.f(d) \preccurlyeq \alpha.g(d')$ pour $\alpha > 0$.
- Stabilité si $\preccurlyeq$ est totale et D est fini : si $\forall d \in D \quad f(d) \preccurlyeq g(d)$ alors $\exists d_1, d_2 \in D \quad \max_{d\in D} f(d) = f(d_1) \preccurlyeq g(d_1) \preccurlyeq g(d_2) = \max_{d\in D} g(d)$.
- Fusion forte finie si $\preccurlyeq$ est totale : $\forall i \in I$ fini $f \preccurlyeq_{\max,D|D_i} g$ ssi $\forall i \in I \quad \exists d_i \in D_i \quad \forall d \in D_i \quad f(d) \preccurlyeq g(d_i)$ alors $\forall d \in \bigcup_{i\in I} D_i \quad f(d) \preccurlyeq \max_{i\in I}(g(d_i))$ lorsque $\preccurlyeq$ est totale et I fini.
- Intersection : si $f\ (\bigcap_{i\in I} \preccurlyeq_i)_{\max}\ g$ alors $\exists d' \in D \quad \forall d \in D \quad \forall i \in I \quad f(d) \preccurlyeq_i g(d')$ donc $\forall i \in I \quad \exists d' \in D \quad \forall d \in D \quad f(d) \preccurlyeq_i g(d')$ donc $\forall i \in I \quad f \preccurlyeq_{i,\max} g$.
- Produit : $(f_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_i)_{\max}\ (g_i)_{i\in I}$ ssi $\exists (d'_i)_{i\in I} \in \prod_{i\in I} D_i \quad \forall (d'_i)_{i\in I} \in \prod_{i\in I} D_i \quad \forall i \in I \quad f_i(d_i) \preccurlyeq_i g_i(d'_i)$.

D.3.6.4 Propriétés préservées par les coûts statiques déduits de coûts statiques généraux.

Coût restreint. [$f \preccurlyeq_{D|D'} g$ ssi $f_{|D'} \preccurlyeq_{D'} g_{|D'}$]

- Reflexivité : $f \preccurlyeq_{D|D'} f$ ssi $f_{|D'} \preccurlyeq_{D'} f_{|D'}$.
- Transitivité : $f \preccurlyeq_{D|D'} g \preccurlyeq_{D|D'} h$ ssi $f_{|D'} \preccurlyeq_{D'} g_{|D'} \preccurlyeq_{D'} h_{|D'}$ donc $f_{|D'} \preccurlyeq_{D'} h_{|D'}$ ssi $f \preccurlyeq_{D|D'} h$.
- Additivité : si $f \preccurlyeq_{D|D'} g$, c.-à-d. $f_{|D'} \preccurlyeq_{D'} g_{|D'}$, alors $f_{|D'} + h_{|D'} \preccurlyeq_{D'} g_{|D'} + h_{|D'}$.
- Homothéticité : si $f \preccurlyeq_{D|D'} g$, c.-à-d. $f_{|D'} \preccurlyeq_{D'} g_{|D'}$, alors $\alpha.f_{|D'} \preccurlyeq_{D'} \alpha.g_{|D'}$ pour $\alpha > 0$ donc $\alpha.f \preccurlyeq_{|D'} \alpha.g$.
- Stabilité : $\preccurlyeq_{\mathrm{abs},D} \subset \preccurlyeq_{\mathrm{abs},D|D'} \subset \preccurlyeq_{D|D'}$ et parce que abs est compatible avec la restriction du support.
- Restriction : c'est l'expression de la définition.
- Fusion forte (resp. faible) : $\forall i \in I \quad f \preccurlyeq_{|D'|D_i} g$ ssi $\forall i \in I \quad f_{|D'\cap D_i} \preccurlyeq_{D'\cap D_i} g_{|D'\cap D_i}$ alors $f_{|D'\cap(\cup_{i\in I} D_i)} \preccurlyeq_{|D'\cap(\cup_{i\in I} D_i)} g_{|D'\cap(\cup_{i\in I} D_i)}$ donc $f \preccurlyeq_{|D'|\cup_{i\in I} D_i} g$.
- Idempotence : $f\ (\preccurlyeq_{|D'})_{|D'}\ g$ ssi $f_{|D'} \preccurlyeq_{D'|D'} g_{|D'}$ ssi $f_{|D'} \preccurlyeq_{D'} g_{|D'}$ ssi $f \preccurlyeq_{|D'} g$.
- Réciproque, complémentaire, relation stricte et équivalence : elles sont sur le modèle de $f\ (\succcurlyeq)_{|D}\ g$ ssi $f_{|D} \succcurlyeq_D g_{|D}$ ssi $f \succcurlyeq_{|D} g$.
- Intersection : $f\ (\bigcap_{i\in I} \preccurlyeq_{i,D})_{|D'}\ g$ ssi $f_{|D'}\ (\bigcap_{i\in I} \preccurlyeq_{i,D'})\ g_{|D'}$ ssi $\forall i \in I \quad f_{|D'} \preccurlyeq_{i,D'} g_{|D'}$ ssi $\forall i \in I \quad f \preccurlyeq_{i,D|D'} g$.
- Produit : $(f_i)_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_{i,D})_{|D'}\ (g_i)_{i\in I}$ ssi $(f_{i|D'})_{i\in I}\ (\bigotimes_{i\in I} \preccurlyeq_{i,D'})\ (g_{i|D'})_{i\in I}$ ssi $\forall i \in I \quad f_{i|D'} \preccurlyeq_{i,D'} g_{i|D'}$ ssi $\forall i \in I \quad f_i \preccurlyeq_{i,D|D'} g_i$.

Coût presque partout. $[\ \preccurlyeq_{\mathrm{pp},D} = \bigcup_{\Delta\,\mathrm{fini}\subset D} \preccurlyeq_{D|D\setminus\Delta}\ ]$

- Réflexivité : elle est préservé par la restriction et la réunion.
- Transitivité : si $f \preccurlyeq_{|D\setminus\Delta} g$ et $g \preccurlyeq_{|D\setminus\Delta'} h$ alors $f \preccurlyeq_{|D\setminus\Delta\cup\Delta'} g \preccurlyeq_{|D\setminus\Delta\cup\Delta'} h$ par restriction et $\Delta \cup \Delta'$ fini.
- Additivité, homothéticité, stabilité : elles sont préservées par le coût restreint et la réunion.
- Restriction : si $D' \subset D$ alors $\preccurlyeq_{\mathrm{pp}(\Delta),D} = \preccurlyeq_{D|D\setminus\Delta} \subset \preccurlyeq_{D|D\setminus\Delta|D'}$ par restriction $= \preccurlyeq_{\mathrm{pp}(\Delta),D|D'}$.
- Fusion forte (respectivement. faible) finie : $\bigcap_{i\in I} \preccurlyeq_{\mathrm{pp}(\Delta_i),D|D_i} = \bigcap_{i\in I} \preccurlyeq_{D|D\setminus\Delta_i|D_i} = \bigcap_{i\in I} \preccurlyeq_{D|D_i\setminus\Delta_i} \subset \preccurlyeq_{D|\cup_{i\in I}(D_i\setminus\Delta_i)}$ par fusion, or $\Delta = (\bigcup_{i\in I} D_i) \setminus (\bigcup_{i\in I}(D_i \setminus \Delta_i)) \subset \bigcup_{i\in I} \Delta_i$ est fini, donc $\preccurlyeq_{D|\cup_{i\in I}(D_i\setminus\Delta_i)} = \preccurlyeq_{\mathrm{pp}(\Delta),D|\cup_{i\in I} D_i}$. Pour la fusion faible, si $\forall i, j \in I \quad i \neq j \Rightarrow D_i \cap D_j = \varnothing$ alors $\forall i, j \in I \quad i \neq j \Rightarrow (D_i \setminus \Delta_i) \cap (D_j \setminus \Delta_j) = \varnothing$.
- Si D est fini : $\preccurlyeq_{\mathrm{pp},D} = \bigcup_{\Delta\,\mathrm{fini}\subset D} \preccurlyeq_{D|D\setminus\Delta} \supset \preccurlyeq_{D|D\setminus\varnothing} = \approx_{\mathrm{triv}}$.
- Idempotence : $(\preccurlyeq_{\mathrm{pp},D})_{\mathrm{pp},D} = \bigcup_{\Delta\,\mathrm{fini}\subset D}(\bigcup_{\Delta'\,\mathrm{fini}\subset D\setminus\Delta} \preccurlyeq_{D|(D\setminus\Delta)\setminus\Delta'}) = \bigcup_{\Delta''\,\mathrm{fini}\subset D} \preccurlyeq_{D|D\setminus\Delta''} = \preccurlyeq_{\mathrm{pp},D}$.
- Réciproque : $(\succcurlyeq)_{\mathrm{pp}(\Delta),D} = (\succcurlyeq)_{|D\setminus\Delta} = \succcurlyeq_{|D\setminus\Delta} = \succcurlyeq_{\mathrm{pp}(\Delta),D}$.
- Complémentaire : $f\ (\not\preccurlyeq)_{\mathrm{pp},D}\ g$ ssi $\exists\Delta$ fini $\subset D \quad f \not\preccurlyeq_{|D\setminus\Delta} g$ si $\forall\Delta$ fini $\subset D \quad f \not\preccurlyeq_{|D\setminus\Delta} g$ ssi $\neg(\exists\Delta$ fini $\subset D \quad f \preccurlyeq_{|D\setminus\Delta} g)$ ssi $f \not\preccurlyeq_{\mathrm{pp},D} g$.
- Intersection (resp. réunion) : comme pour le produit en substituant « $\otimes$ » par « $\cap$ » (resp. « $\cup$ »).
- Produit : $(\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{pp}(\Delta),D} = (\bigotimes_{i\in I} \preccurlyeq_i)_{D|D\setminus\Delta} = \bigotimes_{i\in I} \preccurlyeq_{i,D|D\setminus\Delta} = \bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{pp}(\Delta),D}$. Réciproquement, $\bigotimes_{i\in I} \preccurlyeq_{i,\mathrm{pp}(\Delta_i),D} = \bigotimes_{i\in I} \preccurlyeq_{i,D|D\setminus\Delta_i} \subset \bigotimes_{i\in I} \preccurlyeq_{i,D|D\setminus(\cup_{i\in I}\Delta_i)}$ par restriction $= (\bigotimes_{i\in I} \preccurlyeq_i)_{D|D\setminus(\cup_{i\in I}\Delta_i)} = (\bigotimes_{i\in I} \preccurlyeq_i)_{\mathrm{pp}(\cup_{i\in I}\Delta_i),D}$. Il faut que I soit fini pour garantir que $\bigcup_{j\in I} \Delta_j$ reste aussi fini.

Coût fini. $[\ \preccurlyeq_{\mathrm{fini},D} = \bigcup_{\Delta\,\mathrm{fini}\subset D}(\preccurlyeq_{D|\Delta} \cap \preccurlyeq_{\mathrm{abs},D|D\setminus\Delta})\ ]$

- $(*)$ « si $\Delta \subset \Delta' \subset D$ alors $\preccurlyeq_{\mathrm{fini}(\Delta),D} \subset \preccurlyeq_{\mathrm{fini}(\Delta'),D}$ » : $f \preccurlyeq_{\mathrm{fini}(\Delta),D} g$ ssi $f \preccurlyeq_{D|\Delta} g$ et $f \preccurlyeq_{\mathrm{abs},D|D\setminus\Delta} g$. Puisque abs est compatible avec la restriction du support, on a $f \preccurlyeq_{\mathrm{abs},D|\Delta'\setminus\Delta} g$ et $f \preccurlyeq_{\mathrm{abs},D|D\setminus\Delta'} g$. Puisque $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est stable, on a $f \preccurlyeq_{D|\Delta'\setminus\Delta} g$. Puisque $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ est compatible avec la fusion faible finie des supports, et parce que $\Delta \cap (\Delta' \setminus \Delta) = \varnothing$, on a $f \preccurlyeq_{D|\Delta'} g$. Donc $f \preccurlyeq_{\mathrm{fini}(\Delta'),D} g$.

- Réflexivité : elle est préservé par la restriction, l'intersection et la réunion.
- Antisymétrie : $\preccurlyeq_{\text{fini}(\Delta),D} \cap \succcurlyeq_{\text{fini}(\Delta'),D} \subset \preccurlyeq_{\text{fini}(\Delta\cup\Delta'),D} \cap \succcurlyeq_{\text{fini}(\Delta\cup\Delta'),D}$ par la proposition $(*)$, $= \big(\preccurlyeq_{D|\Delta\cup\Delta'} \cap \preccurlyeq_{\text{abs},D|D\setminus(\Delta\cup\Delta')}\big) \cap \big(\succcurlyeq_{D|\Delta\cup\Delta'} \cap \succcurlyeq_{\text{abs},D|D\setminus(\Delta\cup\Delta')}\big) = \big(\preccurlyeq_{D|\Delta\cup\Delta'} \cap \succcurlyeq_{D|\Delta\cup\Delta'}\big) \cap \big(\preccurlyeq_{\text{abs},D|D\setminus(\Delta\cup\Delta')} \cap \succcurlyeq_{\text{abs},D|D\setminus(\Delta\cup\Delta')}\big) \subset \nabla \cap \nabla = \nabla$.
- Transitivité : si $f \preccurlyeq_{\text{fini}(\Delta),D} g \preccurlyeq_{\text{fini}(\Delta'),D} h$ alors $f \preccurlyeq_{\text{fini}(\Delta\cup\Delta'),D} g \preccurlyeq_{\text{fini}(\Delta\cup\Delta'),D} h$ par $(*)$, ssi $f \preccurlyeq_{D|\Delta\cup\Delta'} g \preccurlyeq_{D|\Delta\cup\Delta'} h$ et $f \preccurlyeq_{\text{abs},D|D\setminus(\Delta\cup\Delta')} g \preccurlyeq_{\text{abs},D|D\setminus(\Delta\cup\Delta')} h$ avec $\Delta \cup \Delta'$ fini.
- Additivité, homothéticité, stabilité, discrimination, compatibilité avec la restriction du support, et composabilité : elles sont vérifiées par le coût absolu et préservées par restriction, intersection et réunion (cf. §3.6.2).
- Fusion forte (resp. faible) finie : $\bigcap_{i\in I} \preccurlyeq_{\text{fini}(\Delta_i),D|D_i} \subset \bigcap_{i\in I} \preccurlyeq_{\text{fini}(\cup_{i\in I}\Delta_i),D|D_i}$ par la proposition $(*)$, $= \bigcap_{i\in I}\big(\preccurlyeq_{D|\cup_{i\in I}\Delta_i|D_i} \cap \preccurlyeq_{\text{abs},D|D\setminus(\cup_{i\in I}\Delta_i)|D_i}\big) = \big(\bigcap_{i\in I} \preccurlyeq_{(\cup_{i\in I}\Delta_i)\cap D_i}\big) \cap \big(\bigcap_{i\in I} \preccurlyeq_{\text{abs},D|D_i\setminus(\cup_{i\in I}\Delta_i)}\big) \subset \preccurlyeq_{|(\cup_{i\in I}\Delta_i)\cap(\cup_{i\in I}D_i)} \cap \preccurlyeq_{\text{abs},D|(\cup_{i\in I}D_i)\setminus(\cup_{i\in I}\Delta_i)} = \preccurlyeq_{\text{fini}(\cup_{i\in I}\Delta_i),D|\cup_{i\in I}D_i}$. On a bien $\bigcup_{i\in I}\Delta_i$ fini lorsque I est fini.
- Si D est fini : $\preccurlyeq_{\text{fini},D} = \bigcup_{\Delta\,\text{fini}\subset D} \preccurlyeq_{\text{fini}(\Delta),D} \supset \preccurlyeq_{\text{fini}(D),D} = \preccurlyeq_D$. Réciproquement $\preccurlyeq_{\text{fini}(\Delta),D} \subset \preccurlyeq_{\text{fini}(D),D}$ par $(*)$, $= \preccurlyeq_{D|D} \cap \preccurlyeq_{\text{abs},D|\varnothing} = \preccurlyeq_D$.
- Idempotence : $(\preccurlyeq_{\text{fini},D})_{\text{fini},D} = \bigcup_{\Delta\,\text{fini}\subset D}\big((\preccurlyeq_{\text{fini},D})_{|\Delta} \cap \preccurlyeq_{\text{abs},D|D\setminus\Delta}\big) = \bigcup_{\Delta\,\text{fini}\subset D}\big(\preccurlyeq_{D|\Delta} \cap \preccurlyeq_{\text{abs},D|D\setminus\Delta}\big) = \preccurlyeq_{\text{fini},D}$.
- Réciproque : elle est déduite de celle pour la coût absolu, pour la restriction, l'intersection et la réunion. $(\succcurlyeq)_{\text{fini}(\Delta),D} = (\succcurlyeq)_{D|\Delta} \cap (\succcurlyeq)_{\text{abs},D|D\setminus\Delta} = \succcurlyeq_{D|\Delta} \cap \succcurlyeq_{\text{abs},D|D\setminus\Delta} = \succcurlyeq_{\text{fini}(\Delta),D}$.
- Complémentaire : $(\not\preccurlyeq)_{\text{fini}(\Delta)} = (\not\preccurlyeq)_{|\Delta} \cap (\not\preccurlyeq)_{\text{abs}|D\setminus\Delta} \subset \not\preccurlyeq_{|\Delta} \cap \not\preccurlyeq_{\text{abs}|D\setminus\Delta} \subset \not\preccurlyeq_{|\Delta} \cup \not\preccurlyeq_{\text{abs}|D\setminus\Delta} = \complement(\preccurlyeq_{|\Delta} \cap \preccurlyeq_{\text{abs}|D\setminus\Delta}) = \not\preccurlyeq_{\text{fini}(\Delta)}$.
- Intersection : comme pour le produit en remplaçant « $\otimes$ » par « $\cap$ ».
- Réunion : $\bigcup_{i\in I} \preccurlyeq_{i,\text{fini}(\Delta_i)} \subset \bigcup_{i\in I} \preccurlyeq_{i,\text{fini}(\cup_{j\in I}\Delta_j)} = \bigcup_{i\in I}\big(\preccurlyeq_{i|\cup_{j\in I}\Delta_j} \cap \preccurlyeq_{i,\text{abs}|D\setminus(\cup_{j\in I}\Delta_j)}\big) \subset \big(\bigcup_{i\in I} \preccurlyeq_{i|\cup_{j\in I}\Delta_j}\big) \cap \big(\bigcup_{i\in I} \preccurlyeq_{i,\text{abs}|D\setminus(\cup_{j\in I}\Delta_j)}\big) = \big(\bigcup_{i\in I} \preccurlyeq_i\big)_{|\cup_{j\in I}\Delta_j} \cap \big(\bigcup_{i\in I} \preccurlyeq_i\big)_{\text{abs}|D\setminus(\cup_{j\in I}\Delta_j)} = \big(\bigcup_{i\in I} \preccurlyeq_i\big)_{\text{fini}(\cup_{j\in I}\Delta_j)}$. Il faut que I soit fini pour garantir que $\bigcup_{j\in I}\Delta_j$ reste aussi fini.
- Compatibilité avec le produit : $\big(\bigotimes_{i\in I} \preccurlyeq_i\big)_{\text{fini}(\Delta)} = \big(\bigotimes_{i\in I} \preccurlyeq_i\big)_{|\Delta} \cap \big(\bigotimes_{i\in I} \preccurlyeq_i\big)_{\text{abs}|D\setminus\Delta} = \bigotimes_{i\in I}\big(\preccurlyeq_{i|\Delta} \cap \preccurlyeq_{i,\text{abs}|D\setminus\Delta}\big) = \bigotimes_{i\in I} \preccurlyeq_{i,\text{fini}(\Delta)}$. Réciproquement, $\bigotimes_{i\in I} \preccurlyeq_{i,\text{fini}(\Delta_i)} \subset \bigotimes_{i\in I} \preccurlyeq_{i,\text{fini}(\cup_{j\in I}\Delta_j)}$ par la proposition $(*)$, $= \bigotimes_{i\in I}\big(\preccurlyeq_{i|\cup_{j\in I}\Delta_j} \cap \preccurlyeq_{i,\text{abs}|D\setminus(\cup_{j\in I}\Delta_j)}\big) = \big(\bigotimes_{i\in I} \preccurlyeq_i\big)_{|\cup_{j\in I}\Delta_j} \cap \big(\bigotimes_{i\in I} \preccurlyeq_i\big)_{\text{abs}|D\setminus(\cup_{j\in I}\Delta_j)} = \big(\bigotimes_{i\in I} \preccurlyeq_i\big)_{\text{fini}(\cup_{j\in I}\Delta_j)}$. Il faut que I soit fini pour garantir que $\bigcup_{j\in I}\Delta_j$ reste aussi fini.

Coût presque fini. **[** $\preccurlyeq_{\text{pfini},D} = \bigcup_{\Delta\,\text{fini}\subset D}\big(\bigcap_{\Delta'\,\text{fini}\supset\Delta} \preccurlyeq_{D|\Delta'}\big)$ **]**

- $(*)$ « si $\Delta_1 \subset \Delta_2$ alors $\preccurlyeq_{\text{pfini}(\Delta_1),D} \subset \preccurlyeq_{\text{pfini}(\Delta_2),D}$ » : $\preccurlyeq_{\text{pfini}(\Delta_1),D} = \bigcap_{\Delta'\,\text{fini}\supset\Delta_1}\big(\preccurlyeq_{D|\Delta'}\big) \subset \bigcap_{\Delta'\,\text{fini}\supset\Delta_2}\big(\preccurlyeq_{D|\Delta'}\big) = \preccurlyeq_{\text{pfini}(\Delta_2),D}$.
- Réflexivité : elle est préservée par la restriction, l'intersection et la réunion (cf. §3.6.2).
- Antisymétrie : $\preccurlyeq_{\text{pfini}(\Delta_1),D} \cap \succcurlyeq_{\text{pfini}(\Delta_2),D} \subset \preccurlyeq_{\text{pfini}(\Delta_1\cup\Delta_2),D} \cap \succcurlyeq_{\text{pfini}(\Delta_1\cup\Delta_2),D}$ par la proposition $(*)$, $= \big(\bigcap_{\Delta'\,\text{fini}\supset\Delta_1\cup\Delta_2} \preccurlyeq_{D|\Delta'}\big) \cap \big(\bigcap_{\Delta'\,\text{fini}\supset\Delta_1\cup\Delta_2} \succcurlyeq_{D|\Delta'}\big) = \bigcap_{\Delta'\,\text{fini}\supset\Delta_1\cup\Delta_2}\big(\preccurlyeq_{D|\Delta'} \cap \succcurlyeq_{D|\Delta'}\big)$ donc si $f \preccurlyeq_{\text{pfini}(\Delta_1),D} g$ et $f \succcurlyeq_{\text{pfini}(\Delta_2),D} g$ alors $\forall d \in D$ $f \preccurlyeq_{D|\Delta_1\cup\Delta_2\cup\{d\}} g$ et $f \succcurlyeq_{D|\Delta_1\cup\Delta_2\cup\{d\}} g$ donc $f_{|\Delta_1\cup\Delta_2\cup\{d\}} = g_{|\Delta_1\cup\Delta_2\cup\{d\}}$ donc $f(d) = g(d)$.
- Transitivité : si $f \preccurlyeq_{\text{pfini}(\Delta_1),D} g \preccurlyeq_{\text{pfini}(\Delta_2),D} h$ alors $f \preccurlyeq_{\text{pfini}(\Delta_1\cup\Delta_2),D} g \preccurlyeq_{\text{pfini}(\Delta_1\cup\Delta_2),D} h$ par $(*)$, ssi $\forall\Delta'$ fini $\supset \Delta_1 \cup \Delta_2$ $f \preccurlyeq_{D|\Delta'} g \preccurlyeq_{D|\Delta'} h$ avec $\Delta_1 \cup \Delta_2$ fini.
- Additivité, homothéticité, stabilité, discrimination et compatibilité avec la restriction du support : elles sont préservées par restriction, intersection et réunion (cf. §3.6.2).
- Fusion forte (resp. faible) finie : $\bigcap_{i\in I} \preccurlyeq_{\text{pfini}(\Delta_i),D|D_i} = \bigcap_{i\in I}\bigcap_{\Delta'\,\text{fini}\supset\Delta_i} \preccurlyeq_{|\Delta'\cap D_i} \subset \bigcap_{i\in I}\bigcap_{\Delta'\,\text{fini}\supset(\cup_{j\in I}\Delta_j)} \preccurlyeq_{|\Delta'\cap D_i} \subset \bigcap_{\Delta'\,\text{fini}\supset(\cup_{j\in I}\Delta_j)} \preccurlyeq_{|\Delta'\cap(\cup_{i\in I}D_i)} = \preccurlyeq_{\text{pfini}(\cup_{i\in I}\Delta_i)|\cup_{i\in I}D_i}$.

- Si D est fini : $\preccurlyeq_{\mathrm{pfini},D} = \bigcup_{\Delta\,\text{fini}\subset D} \preccurlyeq_{\mathrm{pfini}(\Delta),D} = \preccurlyeq_{\mathrm{pfini}(D),D} = \preccurlyeq_D$ par $(*)$.
- Idempotence : $((\preccurlyeq_{\mathrm{pfini},D})_{D\subset\mathcal{D}})_{\mathrm{pfini}(\Delta),D} = \bigcap_{D\supset\Delta'\,\text{fini}\supset\Delta} \preccurlyeq_{\mathrm{pfini},D|\Delta'} = \bigcap_{D\supset\Delta'\,\text{fini}\supset\Delta} \preccurlyeq_{D|\Delta'} = \preccurlyeq_{\mathrm{pfini}(\Delta),D}$.
- Réciproque : elle est déduite de celle pour la restriction, l'intersection et la réunion.
- Complémentaire : $(\not\preccurlyeq)_{\mathrm{pfini}(\Delta)} = \bigcap_{\Delta'\,\text{fini}\supset\Delta}(\not\preccurlyeq)_{|\Delta'} = \bigcap_{\Delta'\,\text{fini}\supset\Delta} \not\preccurlyeq_{|\Delta'} \subset \complement(\bigcap_{\Delta'\,\text{fini}\supset\Delta} \preccurlyeq_{|\Delta'}) = \not\preccurlyeq_{\mathrm{pfini}(\Delta)}$.
- Intersection : comme pour le produit en remplaçant « $\otimes$ » par « $\cap$ ».
- Réunion : $\bigcup_{i\in I} \preccurlyeq_{i,\mathrm{pfini}(\Delta_i)} \subset \bigcup_{i\in I}\preccurlyeq_{i,\mathrm{pfini}(\cup_{j\in I}\Delta_j)}$ par $(*)$, $= \bigcup_{i\in I}(\bigcap_{\Delta'\,\text{fini}\supset(\cup_{j\in I}\Delta_j)} \preccurlyeq_{i|\Delta'}) \subset \bigcap_{\Delta'\,\text{fini}\supset(\cup_{j\in I}\Delta_j)}(\bigcup_{i\in I}\preccurlyeq_{i|\Delta'}) = \bigcap_{\Delta'\,\text{fini}\supset(\cup_{j\in I}\Delta_j)}(\bigcup_{i\in I}\preccurlyeq_i)_{|\Delta'} = (\bigcup_{i\in I}\preccurlyeq_i)_{\mathrm{pfini}(\cup_{j\in I}\Delta_j)}$. Il faut que I soit fini pour garantir que $\bigcup_{j\in I}\Delta_j$ reste aussi fini.
- Produit : $(\bigotimes_{i\in I}\preccurlyeq_i)_{\mathrm{pfini}(\Delta)} = \bigcap_{\Delta'\,\text{fini}\supset\Delta}(\bigotimes_{i\in I}\preccurlyeq_i)_{|\Delta'} = \bigotimes_{i\in I}(\bigcap_{\Delta'\,\text{fini}\supset\Delta}\preccurlyeq_{i|\Delta'}) = \bigotimes_{i\in I}\preccurlyeq_{i,\mathrm{pfini}(\Delta)}$. Réciproquement, $\bigotimes_{i\in I}\preccurlyeq_{i,\mathrm{pfini}(\Delta_i)} \subset \bigotimes_{i\in I}\preccurlyeq_{i,\mathrm{pfini}(\cup_{j\in I}\Delta_j)}$ par $(*)$, $= \bigotimes_{i\in I}(\bigcap_{\Delta'\,\text{fini}\supset(\cup_{j\in I}\Delta_j)}\preccurlyeq_{i|\Delta'}) = \bigcap_{\Delta'\,\text{fini}\supset(\cup_{j\in I}\Delta_j)}(\bigotimes_{i\in I}\preccurlyeq_i) = (\bigotimes_{i\in I}\preccurlyeq_i)_{\mathrm{pfini}(\cup_{j\in I}\Delta_j)}$. Il faut que I soit fini pour garantir que $\bigcup_{j\in I}\Delta_j$ reste aussi fini.

Coût asymptotique local. [$\preccurlyeq_{\mathrm{asymloc},D} = \bigcup_{N\in\mathbb{N}}\bigcap_{n\geqslant N}\preccurlyeq_{D|D_n}$ avec $D_n = \{d\in D\ :\ n = |d|\}$]

- $(*)$ « si $N\leqslant N'$ alors $\preccurlyeq_{\mathrm{asymloc}(N)} \subset \preccurlyeq_{\mathrm{asymloc}(N')}$ » : si $f \preccurlyeq_{\mathrm{asymloc}(N)} g$ alors $\forall n\geqslant N\quad f\preccurlyeq_{D|D_n} g$, or $N\leqslant N'$ donc $\forall n\geqslant N'\quad f\preccurlyeq_{D|D_n} g$.
- Réflexivité, additivité, homothéticité, stabilité et compatibilité avec la réciproque : elles sont préservées par la restriction, la réunion et l'intersection $\preccurlyeq_{\mathrm{asymloc}} = \bigcup_{N\in\mathbb{N}}\bigcap_{n\geqslant N}\preccurlyeq_{D|D_n}$.
- Transitivité : $\preccurlyeq_{\mathrm{asymloc}(N)}\cdot\preccurlyeq_{\mathrm{asymloc}(N')} \subset \preccurlyeq_{\mathrm{asymloc}(\max(N,N'))}\cdot\preccurlyeq_{\mathrm{asymloc}(\max(N,N'))}$ par la proposition $(*)$, $\subset \bigcap_{n\geqslant\max(N,N')}\preccurlyeq_{D|D_n}\cdot\preccurlyeq_{D|D_n} \subset \bigcap_{n\geqslant\max(N,N')}\preccurlyeq_{D|D_n} = \preccurlyeq_{\mathrm{asymloc}(\max(N,N'))}$.
- Fusion forte (respectivement faible) finie : $\bigcap_{i\in I}\preccurlyeq_{\mathrm{asymloc}(N_i)|D'_i} \subset \bigcap_{i\in I}\preccurlyeq_{\mathrm{asymloc}(\max_{i\in I}(N_i))|D'_i}$ par $(*)$, $= \bigcap_{i\in I}\bigcap_{n\geqslant\max_{i\in I}(N_i)}\preccurlyeq_{D|D_n|D'_i} = \bigcap_{n\geqslant\max_{i\in I}(N_i)}\preccurlyeq_{D|D_n\cap(\cup_{i\in I}D'_i)} = \preccurlyeq_{\mathrm{asymloc}(\max_{i\in I}(N_i))|\cup_{i\in I}D'_i}$.
- Si D est fini : l'ensemble des tailles de D est alors borné par un $N\in\mathbb{N}$. Pour tout $n\geqslant N+1$, l'ensemble D_n est vide et donc $\preccurlyeq_{\mathrm{asymloc},D} \cong \preccurlyeq_{D|\varnothing} = \approx_{\mathrm{triv}}$.
- Idempotence : $(\preccurlyeq_{\mathrm{asymloc}})_{\mathrm{asymloc}} = \bigcup_{N\in\mathbb{N}}\bigcap_{n\geqslant N}\preccurlyeq_{\mathrm{asymloc},D|D_n} = \approx_{\mathrm{triv}}$.
- Complémentaire : si $f\ (\not\preccurlyeq)_{\mathrm{asymloc}}\ g$ alors $\exists N\in\mathbb{N}\quad\forall n\geqslant N\quad f\not\preccurlyeq_{|D_n} g$ donc $\forall N'\in\mathbb{N}\quad\exists n'\geqslant N'$ $f\not\preccurlyeq_{|D_{n'}} g$ avec $n'=\max(N,N')$, donc $f\not\preccurlyeq_{\mathrm{asymloc}} g$.
- Intersection : comme pour le produit en remplaçant « $\otimes$ » par « $\cap$ ».
- Réunion : $\bigcup_{i\in I}\preccurlyeq_{i,\mathrm{asymloc}(N_i)} \subset \bigcup_{i\in I}\preccurlyeq_{i,\mathrm{asymloc}(\max_{j\in I}(N_j))}$ par $(*)$, $= \bigcup_{i\in I}(\bigcap_{n\geqslant\max_{j\in I}(N_j)}\preccurlyeq_{i,D|D_n}) \subset \bigcap_{n\geqslant\max_{j\in I}(N_j)}(\bigcup_{i\in I}\preccurlyeq_{i,D|D_n}) = \bigcap_{n\geqslant\max_{j\in I}(N_j)}(\bigcup_{i\in I}\preccurlyeq_{i,D})_{|D_n} = (\bigcup_{i\in I}\preccurlyeq_i)_{\mathrm{asymloc}(\max_{j\in I}(N_j))}$. Il faut que I soit fini pour garantir qu'il existe bien un $\max_{j\in I}(N_j)$.
- Produit : $(\bigotimes_{i\in I}\preccurlyeq_i)_{\mathrm{asymloc}(N)} = \bigcap_{n\geqslant N}(\bigotimes_{i\in I}\preccurlyeq_{i,D|D_n}) = \bigotimes_{i\in I}(\bigcap_{n\geqslant N}\preccurlyeq_{i,D|D_n}) = \bigotimes_{i\in I}\preccurlyeq_{i,\mathrm{asymloc}(N)}$. Réciproquement, $\bigotimes_{i\in I}\preccurlyeq_{i,\mathrm{asymloc}(N_i)} \subset \bigotimes_{i\in I}\preccurlyeq_{i,\mathrm{asymloc}(\max_{j\in I}(N_j))}$ par la prop. $(*)$, $= \bigcap_{n\geqslant\max_{j\in I}(N_j)}(\bigotimes_{i\in I}\preccurlyeq_{i,D|D_n}) = (\bigotimes_{i\in I}\preccurlyeq_i)_{\mathrm{asymloc}(\max_{j\in I}(N_j))}$. Il faut que I soit fini pour garantir qu'il existe bien un $\max_{j\in I}(N_j)$.

Coût asymptotique global. [$\preccurlyeq_{\mathrm{asymglob}} = \bigcup_{N\in\mathbb{N}}\bigcap_{n\geqslant N}\preccurlyeq_{D|D_n}$ avec $D_n = \{d\in D\ :\ n\geqslant |d|\}$]

- Antisymétrie : $\preccurlyeq_{\mathrm{asymglob}(N_1),D}\cap\succcurlyeq_{\mathrm{asymglob}(N_2),D} \subset \preccurlyeq_{\mathrm{asymglob}(\max(N_1,N_2)),D}\cap\succcurlyeq_{\mathrm{asymglob}(\max(N_1,N_2)),D} = \bigcap_{n\geqslant\max(N_1,N_2)}(\preccurlyeq_{D|D_n}\cap\succcurlyeq_{D|D_n})$ donc si $f\preccurlyeq_{\mathrm{asymglob}(N_1),D} g$ et $f\succcurlyeq_{\mathrm{asymglob}(N_2),D} g$ alors $\forall d\in D$ $f\preccurlyeq_{D|D_{\max(N_1,N_2,|d|)}} g$ et $f\succcurlyeq_{D|D_{\max(N_1,N_2,|d|)}} g$ donc $f_{D|D_{\max(N_1,N_2,|d|)}} = g_{D|D_{\max(N_1,N_2,|d|)}}$ donc $f(d)=g(d)$.

- Discrimination : si $f \preccurlyeq_{\mathrm{abs}} g \preccurlyeq_{\mathrm{asymglob}} f$ alors $\exists N \in \mathbb{N} \;\; \forall n \geqslant N \;\; g \preccurlyeq_{D|D_n} f$, donc $g \preccurlyeq_{\mathrm{abs},D|D_n} f$, alors $\forall d_0 \in D \quad g \preccurlyeq_{\mathrm{abs},D|\{d\in D\,:\,\max(|d_0|,N)\geqslant|d|\}} f$, donc $g(d_0) \preccurlyeq f(d_0)$ car $d_0 \in \{d \in D : \max(|d_0|, N) \geqslant |d|\}$, donc $g \preccurlyeq_{D,\mathrm{abs}} f$.
- Si D est fini : l'ensemble des tailles de D est alors borné par un $N \in \mathbb{N}$. Pour tout $n \geqslant N+1$, $D_n = D_N = D$ donc $\preccurlyeq_{\mathrm{asymglob},D} \cong \preccurlyeq_D$.
- Idempotence : $(\preccurlyeq_{\mathrm{asymglob}})_{\mathrm{asymglob}} = \bigcup_{N\in\mathbb{N}}(\bigcap_{n\geqslant N} \preccurlyeq_{\mathrm{asymglob},D|D_n}) = \bigcup_{N\in\mathbb{N}}(\bigcap_{n\geqslant N} \preccurlyeq_{D|D_n}) = \preccurlyeq_{\mathrm{asymglob},D}$.

Le coût asymptotique global diffère du coût local uniquement par le choix de D_n. La démonstration des propriétés restantes est donc identique à celle pour le coût local car elles ne font pas intervenir la nature de D_n.

D.3.6.5 Propriétés préservées par les coûts statiques déduits de coûts statiques.

Coût limite. [$f \preccurlyeq_{\mathrm{lim},D} g$ ssi $\forall\varepsilon > 0 \;\; \exists h \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h\| \leqslant \varepsilon$ et $f \preccurlyeq_D g + h$ avec $(\preccurlyeq_D)_{D\subset\mathcal{D}}$ normé]

Certaines propriétés nécessitent de majorer deux petites variations h_1 et h_2 ; nous avons alors recours à une structure de treillis vectoriel comme celle employée dans la proposition prop. 3.1 (cf. §3.4.7), et l'on définit :

- $\sup_D : \mathcal{C}^D \times \mathcal{C}^D \to \mathcal{C}^D$ tq $\forall f, g \in \mathcal{C}^D \;\; \forall d \in D \;\; \sup_D(f,g)(d) = \sup(f(d), g(d))$

On définit de même $\inf_D$. On a en fait $\sup_D = \sup_{\preccurlyeq_{\mathrm{abs},D}}$. Et $(\mathcal{C}^D, \preccurlyeq_{\mathrm{abs},D}, \sup_D, \inf_D)$ forme un treillis vectoriel.

- Réflexivité : on prend $h = 0$.
- Transitivité : $f_1 \preccurlyeq_{\mathrm{lim},D} f_2 \preccurlyeq_{\mathrm{lim},D} f_3$ ssi $\forall\varepsilon > 0 \;\; \exists h_1, h_2 \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h_1\| \leqslant \varepsilon$ et $\|h_2\| \leqslant \varepsilon$ et $f_1 \preccurlyeq_D f_2 + h_1$ et $f_2 \preccurlyeq_D f_3 + h_2$ alors $f_1 \preccurlyeq_D f_3 + (h_1 + h_2)$ par additivité avec $\|h_1 + h_2\| \leqslant \|h_1\| + \|h_2\| \leqslant 2\varepsilon$.
- Additivité : si $f_1 \preccurlyeq_{\mathrm{lim},D} f_2$ alors $\forall\varepsilon > 0 \;\; \exists h \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h\| \leqslant \varepsilon$ et $f_1 \preccurlyeq_D f_2 + h$, donc $f_1 + g \preccurlyeq_D f_2 + g + h$, donc $f_1 + g \preccurlyeq_{\mathrm{lim},D} f_2 + g$.
- Homothéticité : si $f_1 \preccurlyeq_{\mathrm{lim},D} f_2$ alors $\forall\varepsilon > 0 \;\; \exists h \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h\| \leqslant \varepsilon$ et $f_1 \preccurlyeq_D f_2 + h$, donc $\alpha.f_1 \preccurlyeq_D \alpha.f_2 + \alpha.h$ pour $\alpha > 0$, donc $\alpha.f_1 \preccurlyeq_{\mathrm{lim},D} \alpha.f_2$ car $\|\alpha.h\| \leqslant |\alpha|.\varepsilon$.
- Stabilité : $\preccurlyeq_{\mathrm{abs},D} \subset \preccurlyeq_D \subset \preccurlyeq_{\mathrm{lim},D}$ (avec $h = 0$).
- Discrimination si $\preccurlyeq_D$ est stable : elle est déduite des points (1) et (3) de la proposition 3.2.
- Restriction : si $D' \subset D$ et $\forall\varepsilon > 0 \;\; \exists h \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h\| \leqslant \varepsilon$ et $f \preccurlyeq_D g + h$ alors $f \preccurlyeq_{D|D'} g + h$ donc $f \preccurlyeq_{\mathrm{lim},D|D'} g$ car $\|h_{|D'}\| \leqslant \|h\| \leqslant \varepsilon$.
- Fusion faible finie ; fusion forte finie si $\mathcal{C}^D$ est treillissé : si $\forall i \in I$ fini $f \preccurlyeq_{\mathrm{lim},D|D_i} g$, c.-à-d. $\forall i \in I \;\; \exists h_i \in \mathcal{C}^{D_i}_{\mathrm{norm}} \;\; \|h_i\| \leqslant \varepsilon$ et $f \preccurlyeq_{D|D_i} g + h_i$, alors $\forall i \in I \;\; \exists h'_i \in \mathcal{C}^{D_i}_{\mathrm{norm}} \;\; \|h'_i\| \leqslant \varepsilon$ et $f \preccurlyeq_{D|D_i} g + h'_i$ avec $\forall d \in D_i \;\; h'_i(d) = h_i(d)$ et $\forall d \in D \backslash D_i \;\; h'_i(d) = 0$.
  - Pour la fusion faible, si $\forall i, j \in I \;\; i \neq j \Rightarrow D_i \cap D_j = \varnothing$, alors $\forall i \in I \;\; f \preccurlyeq_{D|D_i} g + h'_i \preccurlyeq_{D|D_1} g + h'$ par stabilité et additivité, avec $h' = \sum_{i\in I} h'_i$.
  - Pour la fusion forte, dans le cas d'un treillis vectoriel, on a $\forall i \in I \;\; f \preccurlyeq_{D|D_i} g + h'_i \preccurlyeq_{D|D_i} g + h'$ par stabilité et additivité, avec $h' = \sup_{D,i\in I}(h'_i)$.

  Dans chaque cas, on a $\|h'\| \preccurlyeq \sum_{i\in I}\|h'_i\| = \sum_{i\in I}\|h_i\| \leqslant |I|\varepsilon$ et $\forall i \in I \;\; f \preccurlyeq_{D|D_i} g + h'$, donc $f \preccurlyeq_{D|\cup_{i\in I} D_i} g + h'$ donc $f \preccurlyeq_{\mathrm{lim},D|\cup_{i\in I} D_i} g$.
- Idempotence : $f \; (\preccurlyeq_{\mathrm{lim},D})_{\mathrm{lim},D} \; g$ ssi $\forall\varepsilon_1 > 0 \;\; \exists h_1 \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h_1\| \leqslant \varepsilon_1$ et $\forall\varepsilon_2 > 0 \;\; \exists h_2 \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h_2\| \leqslant \varepsilon_2$ et $f \preccurlyeq_{\mathrm{lim},D} g + h_2 + h_1$, ssi $f \preccurlyeq_{\mathrm{lim},D} g$ avec $\varepsilon_1 = \varepsilon_2 = \varepsilon$ et $h = h_1 + h_2$.
- Réciproque : $f \; (\succcurlyeq)_{\mathrm{lim},D} \; g$ ssi $\forall\varepsilon > 0 \;\; \exists h \in \mathcal{C}^D_{\mathrm{norm}} \;\; \|h\| \leqslant \varepsilon$ et $f \succcurlyeq_D g + h$ donc $g \preccurlyeq_D f - h$ avec $\|-h\| = \|h\| \leqslant \varepsilon$.

- Intersection (et « $\supset$ » si $\mathcal{C}^D$ est treillissé et I fini) : si $f\ (\bigcap_{i\in I}\preccurlyeq_i)_{\lim,D}\ g$ alors $\forall\varepsilon>0\ \ \exists h\in\mathcal{C}^D_{\text{norm}}$ $\|h\|\leqslant\varepsilon$ et $\forall i\in I\ \ f\preccurlyeq_{i,D} g+h$ donc $\forall i\in I\ \ f\preccurlyeq_{i,\lim,D} g$. Réciproquement, si $\forall i\in I\ \ f\preccurlyeq_{i,\lim,D} g$ alors $\forall\varepsilon>0\ \ \exists(h_i)_{i\in I}\in\mathcal{C}^D_{\text{norm}}\ \ \forall i\in I\ \ \|h_i\|\leqslant\varepsilon$ et $f\preccurlyeq_{i,D} g+h_i$ donc $\forall\varepsilon>0\ \ \exists h\in\mathcal{C}^D_{\text{norm}}$ $\|h\|\leqslant|I|\varepsilon$ et $\forall i\in I\ \ f\preccurlyeq_{i,D} g+h$ par stabilité et additivité, avec $h=\sup_D((h_i)_{i\in I})$ et $\|h\|\leqslant\sum_{i\in I}\|h_i\|\leqslant|I|\varepsilon$. Il faut que I soit fini.
- Réunion (« $\subset$ » si I est fini) : si $f\ (\bigcup_{i\in I}\preccurlyeq_i)_{\lim,D}\ g$ alors $\forall\varepsilon>0\ \ \exists h\in\mathcal{C}^D_{\text{norm}}\ \ \|h\|\leqslant\varepsilon$ et $\exists i_\varepsilon\in I\ \ f\preccurlyeq_{i_\varepsilon,D} g+h$. Si I est fini alors $\exists j\in J\ \ \{j\in I|i_\varepsilon=j\ ,\ \varepsilon>0\}$ est infini et $\forall\varepsilon>0\ \ \exists\, 0<\varepsilon_j\leqslant\varepsilon\ \ i_{\varepsilon_j}=j$. Alors $\forall\varepsilon>0\ \ \exists h\in\mathcal{C}^D_{\text{norm}}\ \ \|h\|\leqslant\varepsilon_j\preccurlyeq\varepsilon$ et $f\preccurlyeq_{i_{\varepsilon_j},D} g+h$, donc $f\preccurlyeq_{j,\lim,D} g$ donc $f\ (\bigcup_{i\in I}\preccurlyeq_{i,\lim,D})\ g$. Réciproquement, si $\exists i\in I\ \ \forall\varepsilon>0\ \ \exists h\in\mathcal{C}^D_{\text{norm}}\ \ \|h\|\leqslant\varepsilon$ et $f\preccurlyeq_{i,D} g+h$ alors $\forall\varepsilon>0\ \ \exists h\in\mathcal{C}^D_{\text{norm}}\ \ \exists i\in I\ \ \|h\|\leqslant\varepsilon$ et $f\preccurlyeq_{i,D} g+h$.
- Produit (« $\supset$ » si I est fini) : $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_i)_{\lim,D}\ (g_i)_{i\in I}$ ssi $\forall\varepsilon>0\ \ \exists(h_i)_{i\in I}\in(\prod_{i\in I}\mathcal{C}_i)^D_{\text{norm}}$ $\|(h_i)_{i\in I}\|=\sum_{i\in I}\|h_i\|_i\preccurlyeq\varepsilon$ et $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_i)\ (g_i)_{i\in I}+(h_i)_{i\in I}$. Donc $\forall j\in I\ \ \forall\varepsilon>0$ $\exists(h_i)_{i\in I}\in(\prod_{i\in I}\mathcal{C}_i)^D_{\text{norm}}\ \ \|h_j\|_j\preccurlyeq\|(h_i)_{i\in I}\|\preccurlyeq\varepsilon$ et $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_i)\ (g_i)_{i\in I}+(h_i)_{i\in I}$, en particulier $f_j\preccurlyeq_j g_j+h_j$, donc $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_{i,\lim,D})\ (g_i)_{i\in I}$. Réciproquement, si $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_{i,\lim,D})$ $(g_i)_{i\in I}$, c.-à-d. $\forall i\in I\ \ \ \forall\varepsilon_i>0\ \ \ \exists h_i\in\mathcal{C}^D_{i,\text{norm}}\ \ \ \|h_i\|_i\preccurlyeq\varepsilon_i$ et $f_i\preccurlyeq_i g_i+h_i$, alors $\forall\varepsilon>0$ $\exists(h_i)_{i\in I}\in(\prod_{i\in I}\mathcal{C}_i)^D_{\text{norm}}\ \ \|(h_i)_{i\in I}\|=\sum_{i\in I}\|h_i\|_i\leqslant\sum_{i\in I}\varepsilon_i\preccurlyeq\varepsilon$ et $\forall i\in I\ \ \ f_i\preccurlyeq_i g_i+h_i$ avec $\varepsilon_i=\varepsilon/|I|$, donc $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_i)_{\lim,D}\ (g_i)_{i\in I}$. Il faut que I soit fini.

Distribution. [$f\preccurlyeq^\nu_D g$ ssi $\nu.f\preccurlyeq_D\nu.g$]

- Réflexivité : $f\preccurlyeq^\nu_D f$ ssi $\nu.f\preccurlyeq_D\nu.f$.
- Antisymétrie : si $\nu.f\preccurlyeq_D\nu.g\preccurlyeq_D\nu.f$ alors $\nu.f=\nu.g$ donc $f=g$si $\nu>0$.
- Transitivité : si $\nu.f\preccurlyeq_D\nu.g\preccurlyeq_D\nu.h$ alors $\nu.f\preccurlyeq_D\nu.h$.
- Totalité : Si $\preccurlyeq_D$ compare tout f et g, elle compare en particulier $\nu.f$ et $\nu.g$.
- Additivité : si $\nu.f\preccurlyeq_D\nu.g$ alors $\nu.f+\nu.h\preccurlyeq_D\nu.g+\nu.h$.
- Homothéticité : si $\nu.f\preccurlyeq_D\nu.g$ alors $\alpha.\nu.f=\nu.\alpha.f\preccurlyeq_D\alpha.\nu.g=\nu.\alpha.g$ pour $\alpha>0$.
- Stabilité : si $\forall d\in D\ \ f(d)\preccurlyeq g(d)$ alors $\forall d\in D\ \ \nu(d).f(d)\preccurlyeq\nu(d).g(d)$, c.-à-d. $\nu.f\preccurlyeq_{\text{abs},D}\nu.g$, donc $\nu.f\preccurlyeq_D\nu.g$.
- Discrimination : si $f\preccurlyeq_{\text{abs},D} g$ et $\nu.g\preccurlyeq_D\nu.f$ alors $\nu.f\preccurlyeq_{\text{abs},D}\nu.g\preccurlyeq_D\nu.f$ donc $\nu.g\preccurlyeq_{\text{abs},D}\nu.f$, donc $g\preccurlyeq_{\text{abs},D} f$ si $\nu>0$.
- Restriction : si $\nu.f\preccurlyeq_D\nu.g$ alors $\forall D'\subset D\ \ \nu.f\preccurlyeq_{D|D'}\nu.g$.
- Fusion forte (resp. faible) : si $\forall i\in I\ \ \nu.f\preccurlyeq_{D|D_i}\nu.g$ alors $\forall i\in I\ \ \nu.f\preccurlyeq_{D|\cup_{i\in I}}\nu.g$.
- Composabilité à droite : Si $f\preccurlyeq^\nu_D f'$, c.-à-d. $\nu.f\preccurlyeq_D\nu.f'$ alors $\nu.h+\nu.f\circ g\preccurlyeq_{g^{-1}(D)}\nu.h+\nu.f'\circ g$ donc $h+f\circ g\preccurlyeq^\nu_{g^{-1}(D)} h+f'\circ g$.
- Réciproque : $f\ (\succcurlyeq)^\nu\ g$ ssi $\nu.f\succcurlyeq\nu.g$ ssi $f\succcurlyeq^\nu g$.
- Complémentaire : $f\ (\not\preccurlyeq)^\nu\ g$ ssi $\nu.f\not\preccurlyeq\nu.g$ ssi $f\not\preccurlyeq^\nu g$.
- Intersection : $f\ (\bigcap_{i\in I}\preccurlyeq_{i,D})^\nu\ g$ ssi $\forall i\in I\ \ \nu.f\preccurlyeq_{i,D}\nu.g$ ssi $f\ (\bigcap_{i\in I}\preccurlyeq_{i,D}{}^\nu)\ g$.
- Réunion : par l'intersection et le complémentaire.
- Produit : $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_{i,D})^\nu\ (g_i)_{i\in I}$ ssi $\forall i\in I\ \ \nu.f_i\preccurlyeq_{i,D}\nu.g_i$ ssi $(f_i)_{i\in I}\ (\otimes_{i\in I}\preccurlyeq_{i,D}{}^\nu)\ (g_i)_{i\in I}$.

Coût majoré d'un facteur au moins $\alpha_0>1$. [$f\preccurlyeq_{\text{maj}(\alpha_0),D} g$ ssi $\forall\alpha\in A\ \ \alpha.f\preccurlyeq_D g$ avec $A=[0,\alpha_0]$]

- Antisymétrie (triviale) : par inclusion (tabl. 3.2) car $\preccurlyeq_{\text{maj}(A),D}=\bigcap_{\alpha\in A}\preccurlyeq_{\text{maj}\{\alpha\},D}\subset\preccurlyeq_{\text{maj}\{1\},D}=\preccurlyeq_D$. Voir aussi le cas de l'équivalence ci-dessous.

- Transitivité : si $\forall\alpha\in A \quad \alpha.f \preccurlyeq g$ et $\forall\alpha\in A \quad \alpha.g \preccurlyeq h$ alors $\forall\alpha\in A \quad \sqrt{\alpha}.f \preccurlyeq g$ et $\sqrt{\alpha}.g \preccurlyeq h$ car $\sqrt{\alpha}\in A$. En effet si $\alpha\leqslant 1$ alors $\sqrt{\alpha}\leqslant 1<\alpha_0$ et si $\alpha>1$ alors $\sqrt{\alpha}\leqslant\sqrt{\alpha_0}<\alpha_0$. Donc $\alpha.f=(\sqrt{\alpha})^2.f \preccurlyeq \sqrt{\alpha}.g \preccurlyeq h$ par homothéticité.
- Homothéticité : si $\forall\alpha\in A \quad \alpha.f \preccurlyeq_D g$ alors $\forall\alpha\in A \quad \alpha.\alpha'.f=\alpha'.\alpha.f \preccurlyeq \alpha'.g$ pour $\alpha'>0$.
- Discrimination : par inclusion (tabl. 3.2) car $\preccurlyeq_{\mathrm{maj}(A),D}\subset\preccurlyeq_D$.
- Restriction : si $\forall\alpha\in A \quad \alpha.f \preccurlyeq_D g$ alors $\forall D'\subset D \quad \forall\alpha\in A \quad \alpha.f \preccurlyeq_{D|D'} g$.
- Fusion forte (resp. faible) : si $\forall\alpha\in A \quad \forall i\in I \quad \alpha.f \preccurlyeq_{D|D_i} g$ alors $\forall i\in I \quad \forall\alpha\in A \quad \alpha.f \preccurlyeq_{D|\cup_{i\in I}D_i} g$.
- Idempotence : $(\preccurlyeq_{\mathrm{maj}(\alpha)})_{\mathrm{maj}(\alpha)}=\preccurlyeq_{\mathrm{maj}(\alpha^2)}\subset\preccurlyeq_{\mathrm{maj}(\alpha)}$ car $1<\alpha<\alpha^2$ (cf. §3.6.6, Coûts majorés).
- Complémentaire : si $f\ (\not\preccurlyeq)_{\mathrm{maj}(A)}\ g$, c.-à-d. $\forall\alpha\in A \quad \alpha.f \not\preccurlyeq g$, alors $\exists\alpha\in A \quad \alpha.f \not\preccurlyeq g$, donc $\neg(\forall\alpha\in A \quad \alpha.f \not\preccurlyeq g)$, c.-à-d. $f \not\preccurlyeq_{\mathrm{maj}(A)} g$.
- Relation stricte : si $f\ (\prec)_{\mathrm{maj}(A)}\ g$, c.-à-d. $\forall\alpha\in A \quad \alpha.f \preccurlyeq g$ et $\alpha.f \not\succcurlyeq g$ alors $\forall\alpha\in A \quad \alpha.f \preccurlyeq g$ et $\exists\, 1<\alpha'\in A \quad \alpha'.f \not\succcurlyeq g$, donc $\forall\alpha\in A \quad \alpha.f \preccurlyeq g$ et $\exists\, 1<\alpha'\in A \quad f \not\succcurlyeq (1/\alpha').g$ par homothéticité, donc $\forall\alpha\in A \quad \alpha.f \preccurlyeq g$ et $\exists\alpha'\in A \quad f \not\succcurlyeq \alpha'.g$ car $1/\alpha'<1$, donc $\forall\alpha\in A \quad \alpha.f \preccurlyeq g$ et $\neg(\forall\alpha'\in A \quad f \succcurlyeq \alpha'.g)$, donc $f \preccurlyeq_{\mathrm{maj}(A)} g$ et $\neg(f \succcurlyeq_{\mathrm{maj}(A)} g)$, c.-à-d. $f \prec_{\mathrm{maj}(A)} g$.
- Équivalence (triviale) : supposons $\forall\alpha\in A \quad \alpha.f \preccurlyeq g$ et $\forall\alpha\in A \quad \alpha.g \preccurlyeq f$. Soit $1<\alpha<\alpha_0$, alors $\alpha^2.f \preccurlyeq f$ par homothéticité, donc $(\alpha^2-1).f \preccurlyeq 0$ par additivité, donc $f \preccurlyeq 0$ par homothéticité. Or $0=0.g \preccurlyeq f$ donc $f\approx 0$. De même $g\approx 0$, donc $\forall\alpha\in A \quad \alpha.f \approx g$, donc $\approx_{\mathrm{maj}(A)}\subset(\approx)_{\mathrm{maj}(A)}$. Réciproquement si $\forall\alpha\in A \quad \alpha.f \approx g$ alors $0=0.f\approx g$ et $f\approx(1/\alpha)g\approx 0$, donc $\forall\alpha\in A$ $\alpha.f \preccurlyeq g$ et $\forall\alpha\in A \quad \alpha.g \preccurlyeq f$, donc $(\approx)_{\mathrm{maj}(A)}\subset\approx_{\mathrm{maj}(A)}$.
- Intersection : $f\ (\bigcap_{i\in I}\preccurlyeq_{i,D})_{\mathrm{maj}(\alpha)}\ g$ ssi $\forall i\in I \quad \forall\alpha\in A \quad \alpha.f \preccurlyeq_{i,D} g$ ssi $f\ (\bigcap_{i\in I}\preccurlyeq_{i,D_{\mathrm{maj}(\alpha)}})\ g$.
- Réunion : $f\ (\bigcup_{i\in I}\preccurlyeq_{i,D})_{\mathrm{maj}(\alpha)}\ g$ ssi $\forall\alpha\in A \quad \exists i\in I \quad \alpha.f \preccurlyeq_{i,D} g$ si $\exists i\in I \quad \forall\alpha\in A \quad \alpha.f \preccurlyeq_{i,D} g$ ssi $f\ (\bigcup_{i\in I}\preccurlyeq_{i,D_{\mathrm{maj}(\alpha)}})\ g$.
- Produit : $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_{i,D})_{\mathrm{maj}(\alpha)}\ (g_i)_{i\in I}$ ssi $\forall\alpha\in A \quad \forall i\in I \quad \alpha.f_i \preccurlyeq_{i,D} g_i$ ssi $(f_i)_{i\in I}\ (\bigotimes_{i\in I}\preccurlyeq_{i,D_{\mathrm{maj}(\alpha)}})\ (g_i)_{i\in I}$.

Coût majoré d'un facteur limite au moins $\alpha_0>1$. [$f \preccurlyeq_{\mathrm{majl}(\alpha_0),D} g$ ssi $\forall\alpha\in[0,\alpha_0[\quad \alpha.f \preccurlyeq_D g$]

Il suffit de remplacer $A=[0,\alpha_0]$ par $A=[0,\alpha_0[$ dans les démonstrations précédentes pour le *coût majoré d'un facteur au moins* $\alpha_0>1$, car nous n'avons jamais employé l'hypothèse $\alpha_0\in A$. Alternativement, on peut aussi déduire certaines propriétés par compatibilité avec la réunion car $\preccurlyeq_{\mathrm{majl}(\alpha_0)}=\bigcup_{1<\alpha<\alpha_0}\preccurlyeq_{\mathrm{maj}(\alpha)}$.

Coût majoré. [$\preccurlyeq_{\mathrm{maj}}=\bigcup_{1<\alpha}\preccurlyeq_{\mathrm{maj}(\alpha)}$]

- Antisymétrie (triviale) : si $f \preccurlyeq_{\mathrm{maj}(\alpha_1)} g$ et $g \preccurlyeq_{\mathrm{maj}(\alpha_2)} f$ alors $f \preccurlyeq_{\mathrm{maj}(\min(\alpha_1,\alpha_2))} g$ et $g \preccurlyeq_{\mathrm{maj}(\min(\alpha_1,\alpha_2))} f$ lorsque $\preccurlyeq_D$ est homothétique, avec $1<\min(\alpha_1,\alpha_2)$, et l'on est ramené au cas de $\preccurlyeq_{\mathrm{maj}(\alpha)}$. Voir aussi cas de l'équivalence ci-dessous.
- Transitivité : si $f \preccurlyeq_{\mathrm{maj}(\alpha_1)} g$ et $g \preccurlyeq_{\mathrm{maj}(\alpha_2)} h$ alors $f \preccurlyeq_{\mathrm{maj}(\min(\alpha_1,\alpha_2))} g$ et $g \preccurlyeq_{\mathrm{maj}(\min(\alpha_1,\alpha_2))} h$ lorsque $\preccurlyeq_D$ est homothétique, avec $1<\min(\alpha_1,\alpha_2)$.
- Homothéticité, discrimination, restriction et compatibilité avec la réciproque et la réunion : elles sont préservées par la réunion $\preccurlyeq_{\mathrm{maj}}=\bigcup_{1<\alpha}\preccurlyeq_{\mathrm{maj}(\alpha)}$.
- Fusion forte (resp. faible) finie : Si I est fini alors $\bigcap_{i\in I}\preccurlyeq_{\mathrm{maj}(\alpha_i),D|D_i}\subset\bigcap_{i\in I}\preccurlyeq_{\mathrm{maj}(\min_{j\in I}(\alpha_j)),D|D_i}\subset\preccurlyeq_{\mathrm{maj}(\min_{j\in I}(\alpha_j)),D|\cup_{i\in I}D_i}$ avec $\min_{j\in I}(\alpha_j)>1$.
- Idempotence : $(\preccurlyeq_{\mathrm{maj}})_{\mathrm{maj}}=\bigcup_{1<\alpha}(\bigcup_{1<\alpha'}\preccurlyeq_{\mathrm{maj}(\alpha')})_{\mathrm{maj}(\alpha)}=\bigcup_{1<\alpha,\alpha'}\preccurlyeq_{\mathrm{maj}(\alpha\alpha')}=\preccurlyeq_{\mathrm{maj}}$.

- Relation stricte : si $f\ (\prec)_{\mathrm{maj}}\ g$, c.-à-d. $\exists\alpha>1\quad f\ (\prec)_{\mathrm{maj}(\alpha)}\ g$ alors $f\prec_{\mathrm{maj}(\alpha)} g$, donc $f\prec_{\mathrm{maj}} g$.
- Équivalence (triviale) : si $f\preccurlyeq_{\mathrm{maj}} g$ et $g\preccurlyeq_{\mathrm{maj}} f$ alors $\exists\alpha_1>1\quad\exists\alpha_2>1\quad f\preccurlyeq_{\mathrm{maj}(\alpha_1)} g$ et $g\preccurlyeq_{\mathrm{maj}(\alpha_2)} f$ alors $f\preccurlyeq_{\mathrm{maj}(\alpha)} g$ et $g\preccurlyeq_{\mathrm{maj}(\alpha)} f$ avec $\alpha=\min(\alpha_1,\alpha_2)>1$, donc $f\approx g\approx 0$ car $\approx_{\mathrm{maj}(\alpha)}=\{(0,0)\}$.
- Intersection : $(\bigcap_{i\in I}\preccurlyeq_i)_{\mathrm{maj}(\alpha)}=\bigcap_{i\in I}\preccurlyeq_{i,\mathrm{maj}(\alpha)}$. On a réciproquement, $\bigcap_{i\in I}\preccurlyeq_{i,\mathrm{maj}(\alpha_i)}\subset\bigcap_{i\in I}\preccurlyeq_{i,\mathrm{maj}(\min_{j\in I}(\alpha_j))}=(\bigcap_{i\in I}\preccurlyeq_i)_{\mathrm{maj}(\min_{j\in I}(\alpha_j))}$. Il faut que I soit fini pour qu'il existe $\min_{j\in I}(\alpha_j)>1$.
- Produit : Comme l'intersection en remplaçant « $\cap$ » par « $\otimes$ ».

Coût négligeable. [$\preccurlyeq_{\text{négl}}=\bigcap_{0\leqslant\alpha}\preccurlyeq_{\mathrm{maj}(\alpha)}$]

Il suffit de remplacer $A=[0,\alpha_0]$ par $A=[0,+\infty[$ dans les démonstrations précédentes pour le *coût majoré d'un facteur au moins* $\alpha_0>1$, car nous n'avons jamais employé l'hypothèse que A était borné. Alternativement, on peut aussi déduire certaines propriétés par compatibilité avec l'intersection car $\preccurlyeq_{\text{négl}}=\bigcap_{0\leqslant\alpha}\preccurlyeq_{\mathrm{maj}(\alpha)}$.

- Idempotence : $(\preccurlyeq_{\text{négl}})_{\text{négl}}=\bigcap_{0\leqslant\alpha}(\bigcap_{0\leqslant\alpha'}\preccurlyeq_{\mathrm{maj}(\alpha')})_{\mathrm{maj}(\alpha)}=\bigcap_{0\leqslant\alpha,\alpha'}\preccurlyeq_{\mathrm{maj}(\alpha\alpha')}=\preccurlyeq_{\text{négl}}$.

D.3.6.6 Graduation des coûts.

Coût statique déduit d'un coût statique général.

(1) • $\preccurlyeq_{\mathrm{fini}(\Delta),D}=\preccurlyeq_{D|\Delta}\cap\preccurlyeq_{\mathrm{abs},D|D\setminus\Delta}=\preccurlyeq_{D|\Delta}\cap\preccurlyeq_{\text{abs-pp}(\Delta),D}\subset\preccurlyeq_{\text{abs-pp}(\Delta),D}$.

• $\preccurlyeq_{\text{abs-pp}(\Delta),D}=\preccurlyeq_{\mathrm{abs},D|D\setminus\Delta}\subset\preccurlyeq_{D|D\setminus\Delta}=\preccurlyeq_{\mathrm{pp}(\Delta),D}$ par stabilité.

• Si $f\preccurlyeq_{\mathrm{pp}(\Delta),D} g$, soit $N=\max_{d\in\Delta}(|d|)$, alors $\forall n>N\quad f\preccurlyeq_{D|D_n} g$ avec $D_n=\{d\in D\mid n=|d|\}$, donc $f\preccurlyeq_{\mathrm{asymloc},D} g$.

(2) • $\preccurlyeq_{\mathrm{fini}(\Delta),D}=\preccurlyeq_{D|\Delta}\cap\preccurlyeq_{\mathrm{abs},D|D\setminus\Delta}\subset\preccurlyeq_{D|\Delta}\cap(\bigcap_{\Delta'\ \mathrm{fini}\subset D\setminus\Delta}\preccurlyeq_{\mathrm{abs},D|\Delta'})\subset\preccurlyeq_{D|\Delta}\cap(\bigcap_{\Delta'\ \mathrm{fini}\subset D\setminus\Delta}\preccurlyeq_{D|\Delta'})$ par stabilité, $=\bigcap_{\Delta'\,\mathrm{fini}\subset D\setminus\Delta}(\preccurlyeq_{D|\Delta}\cap\preccurlyeq_{D|\Delta'})\subset\bigcap_{\Delta'\ \mathrm{fini}\supset\Delta}\preccurlyeq_{D|\Delta'}$ par fusion faible, $=\preccurlyeq_{\mathrm{pfini},D}$.

• Si $f\preccurlyeq_{\mathrm{pfini}(\Delta),D} g$ alors $\forall\Delta'$ fini $\supset\Delta\quad f\preccurlyeq_{D|\Delta'} g$. Soient $N=\max_{d\in\Delta}(|d|)$ et $\forall n\in\mathbb{N}\quad D_n=\{d\in D\mid n\geqslant|d|\}$, alors $\forall n\geqslant N\quad D_n$ fini $\supset D_N\supset\Delta$ donc $f\preccurlyeq_{D|D_n} g$, c.-à-d. $f\preccurlyeq_{\mathrm{asymglob},D} g$.

• $\preccurlyeq_{\mathrm{asymglob}(N)}=\bigcap_{n\geqslant N}\preccurlyeq_{|\{d\in D\,:\,n\geqslant|d|\}}\subset\bigcap_{n\geqslant N}\preccurlyeq_{|\{d\in D\,:\,n=|d|\}}$ par compatibilité avec la restriction, $=\preccurlyeq_{\mathrm{asymloc}(N)}$.

Distribution.

(3) $f\ (\preccurlyeq^{\nu_1})^{\nu_2}\ g$ ssi $\nu_2.f\preccurlyeq^{\nu_1}\nu_2.g$ ssi $\nu_1.\nu_2.f\preccurlyeq\nu_1.\nu_2.g$.

(4) Si $\forall d\in D\quad f(d)\preccurlyeq g(d)$ alors $\forall d\in D\quad \nu(d).f(d)\preccurlyeq\nu(d).g(d)$. La réciproque est est vraie si $\forall d\in D$ $\nu(d)\neq 0$. Le raisonnement est identique pour abs-pp, sur l'ensemble $D\setminus\Delta$ avec Δ fini.

(5) $f\ (\preccurlyeq_{|D'})^{\nu}\ g$ ssi $\nu.f\preccurlyeq_{|D'}\nu.g$ ssi $(\nu.f)_{|D'}\preccurlyeq_{D'}(\nu.g)_{|D'}$ ssi $f_{|D'}\preccurlyeq^{\nu} g_{|D'}$ ssi $f\ (\preccurlyeq^{\nu})_{|D'}\ g$.

Coût limite.

(6) Si $f\preccurlyeq_D g$ alors $f\preccurlyeq_{\mathrm{lim},D} g+h$ avec $h=0\in\mathcal{C}^D_{\mathrm{norm}}$.

Coût maximum.

(7) Si $\exists d'\in D\quad\forall d\in D\quad f(d)\leqslant g(d')$ alors $\forall n\geqslant|d'|\quad\forall d\in D_n\quad f(d)\leqslant g(d')$, avec $D_n=\{d\in D\mid n\leqslant|d|\}$ et $d'\in D_n$.

(8) • Si $\forall d \in D \quad f(d) \preccurlyeq g(d)$ alors $\forall n \in \mathrm{IN} \quad \exists d' \in D_n \quad \forall d \in D_n \quad f(d) \preccurlyeq g(d')$, avec $g(d') = \max_{d \in D_n} g(d)$ et $D_n = \{d \in D \mid n \leqslant |d|\}$, donc $\preccurlyeq_{\text{abs}} \subset \preccurlyeq_{\text{max-asymloc}}$.
• Le point précédent s'applique aussi à $\preccurlyeq_{\text{max-asymglob}}$ avec $D_n = \{d \in D \mid n = |d|\}$.

Coûts majorés. Notons $(*)$ tout d'abord que « $(\preccurlyeq_{\text{maj}\{\alpha_1\}})_{\text{maj}\{\alpha_2\}} = \preccurlyeq_{\text{maj}\{\alpha_1\alpha_2\}}$ ». En effet, pour tout $f, g \in \mathcal{C}^D$ on a $f\ (\preccurlyeq_{\text{maj}\{\alpha_1\}})_{\text{maj}\{\alpha_2\}}\ g$ ssi $\alpha_2.f \preccurlyeq_{\text{maj}\{\alpha_1\}} g$ ssi $\alpha_1\alpha_2.f \preccurlyeq g$ ssi $f \preccurlyeq_{\text{maj}\{\alpha_1\alpha_2\}} g$.

(9) $(\preccurlyeq_{\text{maj}(\alpha_1)})_{\text{maj}(\alpha_2)} = \bigcap_{0 \leqslant \alpha'_2 \leqslant \alpha_2} (\bigcap_{0 \leqslant \alpha'_1 \leqslant \alpha_1} \preccurlyeq_{\text{maj}\{\alpha'_1\}})_{\text{maj}\{\alpha'_2\}} = \bigcap_{0 \leqslant \alpha'_2 \leqslant \alpha_2} \bigcap_{0 \leqslant \alpha'_1 \leqslant \alpha_1} \preccurlyeq_{\text{maj}\{\alpha'_1\alpha'_2\}}$ par $(*)$, $= \bigcap_{0 \leqslant \alpha \leqslant \alpha_1\alpha_2} \preccurlyeq_{\text{maj}\{\alpha\}} = \preccurlyeq_{\text{maj}(\alpha_1\alpha_2)}$.

(10) $(\preccurlyeq_{\text{majl}(\alpha_1)})_{\text{maj}(\alpha_2)} = \bigcap_{0 \leqslant \alpha'_2 < \alpha_2} (\bigcap_{0 \leqslant \alpha'_1 \leqslant \alpha_1} \preccurlyeq_{\text{maj}\{\alpha'_1\}})_{\text{maj}\{\alpha'_2\}} = \bigcap_{0 \leqslant \alpha'_2 < \alpha_2} \bigcap_{0 \leqslant \alpha'_1 \leqslant \alpha_1} \preccurlyeq_{\text{maj}\{\alpha'_1\alpha'_2\}}$ par $(*)$, $= \bigcap_{0 \leqslant \alpha < \alpha_1\alpha_2} \preccurlyeq_{\text{maj}\{\alpha\}} = \preccurlyeq_{\text{majl}(\alpha_1\alpha_2)}$. Le raisonnement est identique pour les autres égalités.

(11) si $\forall \alpha'_1 \leqslant \alpha_1 \quad \alpha'_1.f \preccurlyeq g$ et $\forall \alpha'_2 \leqslant \alpha_2 \quad \alpha'_2.g \preccurlyeq h$ alors $\forall \alpha \leqslant \alpha_1\alpha_2 \quad \exists \alpha'_1 \leqslant \alpha_1 \quad \exists \alpha'_2 \leqslant \alpha_2 \quad \alpha = \alpha'_1\alpha'_2$ et $\alpha'_1.f \preccurlyeq g$ et $\alpha'_2.g \preccurlyeq h$, alors $\alpha'_1\alpha'_2.f \preccurlyeq \alpha'_2.g \preccurlyeq h$ par homothéticité, donc $\forall \alpha \leqslant \alpha_1\alpha_2 \quad \alpha.f \preccurlyeq h$.

(12) $\preccurlyeq_{\text{maj}(\alpha_1)} \cdot \preccurlyeq_{\text{majl}(\alpha_2)} = \preccurlyeq_{\text{maj}(\alpha_1)} \cdot (\bigcap_{0 \leqslant \alpha < \alpha_2} \preccurlyeq_{\text{maj}(\alpha)} \subset \bigcap_{0 \leqslant \alpha < \alpha_2} (\preccurlyeq_{\text{maj}(\alpha_1)} \cdot \preccurlyeq_{\text{maj}(\alpha)}) \subset \bigcap_{0 \leqslant \alpha < \alpha_2} \preccurlyeq_{\text{maj}(\alpha_1\alpha)}$ $= \bigcap_{0 \leqslant \alpha < \alpha_1\alpha_2} \preccurlyeq_{\text{maj}(\alpha)} = \preccurlyeq_{\text{majl}(\alpha_1\alpha_2)}$. Le raisonnement est identique pour les autres inclusions. Alternativement, on peut simplement reprendre le point précédent avec des « $<$ » à la place de certains « $\leqslant$ ».

(13) $\preccurlyeq_{\text{négl}} = \bigcap_{0 \leqslant \alpha} \preccurlyeq_{\text{maj}\{\alpha\}} \subset \bigcap_{0 \leqslant \alpha \preccurlyeq \alpha_2} \preccurlyeq_{\text{maj}\{\alpha\}} = \preccurlyeq_{\text{maj}(\alpha_2)} \subset \bigcap_{0 \leqslant \alpha \prec \alpha_2} \preccurlyeq_{\text{maj}\{\alpha\}} = \preccurlyeq_{\text{majl}(\alpha_2)} \subset \bigcap_{0 \leqslant \alpha \preccurlyeq \alpha_1} \preccurlyeq_{\text{maj}\{\alpha\}} = \preccurlyeq_{\text{maj}(\alpha_1)} \subset \bigcap_{0 \leqslant \alpha \prec \alpha_1} \preccurlyeq_{\text{maj}\{\alpha\}} = \preccurlyeq_{\text{majl}(\alpha_1)} \subset \bigcup_{1 < \alpha} \preccurlyeq_{\text{maj}(\alpha)} = \preccurlyeq_{\text{maj}} \subset \preccurlyeq_{\text{maj}\{1\}} = \preccurlyeq_D$.

(14) L'identité est déduite de $\preccurlyeq_{\text{maj}(\alpha_2)} \subset \preccurlyeq_{\text{maj}(\alpha_1)}$ lorsque $\alpha_1 < \alpha_2$ (pt. 13).

(15) L'identité est déduite de $\preccurlyeq_{\text{majl}(\alpha_2)} \subset \preccurlyeq_{\text{majl}(\alpha_1)}$ lorsque $\alpha_1 < \alpha_2$ (pt. 13)).

Support fini.

(16) • $\preccurlyeq_{\text{abs}} \subset \preccurlyeq_{\text{moy}}$ est la condition de stabilité (cf. §3.6.3).
• Supposons $f \preccurlyeq_{\text{moy-lim}} g$ alors $\forall \varepsilon > 0 \quad \exists c \in \mathcal{C} \quad \|c\| \leqslant \varepsilon$ et $\sum_{d \in D} f(d) \leqslant \sum_{d \in D} g(d) + c$ car D est fini. Soit $c' = \sum_{d \in D} g(d) - f(d)$. Considérons $\varepsilon = 1/(n+1)$ pour $n \in \mathrm{IN}$. Il existe alors une suite $(c_n)_{n \in \mathrm{IN}}$ de $\mathcal{C}_+$ vérifiant : $\forall n \in \mathrm{IN} \quad \|c_n\| < 1/(n+1)$ et $0 \leqslant c' + c_n$. La suite $(c'_n)_{n \in \mathrm{IN}} = (c' + c_n)_{n \in \mathrm{IN}}$ converge vers c' en restant dans le fermé $\mathcal{C}_+$, donc $c' \geqslant 0$, c'est-à-dire $f \preccurlyeq_{\text{moy}} g$. Autrement dit, $\preccurlyeq_{\text{moy-lim}} \subset \preccurlyeq_{\text{lim}}$. L'inclusion réciproque est donnée par le point (6).

(17) • $\preccurlyeq_{\text{abs}} \subset \preccurlyeq_{\text{max}}$ est la condition de stabilité (cf. §3.6.3).
• Supposons $f \preccurlyeq_{\text{max-lim}} g$ alors $\forall \varepsilon > 0 \quad \exists c \in \mathcal{C} \quad \|c\| \leqslant \varepsilon$ et $\exists d' \in D \quad \forall d \in D \quad f(d) \preccurlyeq g(d') + c$ car D est fini. Soit $c' = g(d') - f(d)$. Le même raisonnement qu'au point (16) ci-dessus conduit à $c' \succcurlyeq 0$, c'est-à-dire $f \preccurlyeq_{\text{max}} g$. Autrement dit, $\preccurlyeq_{\text{max-lim}} \subset \preccurlyeq_{\text{lim}}$. L'inclusion réciproque est aussi donnée par le point (6).

(18) $\preccurlyeq_{\text{pp}(D)} = \preccurlyeq_{|\varnothing} = \approx_{\text{triv}}$ et $\preccurlyeq_{\text{asymloc}} \supset \preccurlyeq_{\text{asymloc}(1+\max_{deD}|d|)} = \preccurlyeq_{|\varnothing} = \approx_{\text{triv}}$.

(19) Ces points figurent dans les tableaux 3.4 et 3.5.

Coûts d'usage pratique.

- Les programmes `double1` et `double2` (cf. §3.4.2) montrent que `double1` $\not\preccurlyeq_{\text{abs}}$ `double2` et `double1` $\succcurlyeq_{\text{moy-fini}}$ `double2`.
- Les deux fonctions f et g données en exemple du coût moyen presque fini (cf. §3.4.6) vérifient $f \preccurlyeq_{\text{moy-pfini}} g$ et $f \not\preccurlyeq_{\text{moy-fini}} g$ car f n'est inférieure à g que sur $\{0\}$.
- Le même exemple vérifie $g \preccurlyeq_{\text{moy-lim}} f$ et $g \not\preccurlyeq_{\text{moy-pfini}} f$.
- On a, avec D singleton, $\langle 2 \rangle \not\preccurlyeq_{\text{moy-fini}} \langle 1 \rangle$ mais $\langle 2 \rangle \preccurlyeq_{\text{abs-pp}} \langle 1 \rangle$.

- Soient f et g définies par $f(n) = (-1)^n$ et $g(n) = (-1)^{n+1}$ pour $n \in \mathbb{N}$, ainsi qu'une taille $|n| = \lfloor n/2 \rfloor$. Alors $f \approx_{\text{moy-asymglob}} g$, mais f et g ne sont pas comparables par $\preccurlyeq_{\text{moy-pfini}}$.

D.4 Optimisation et optimalité.

Démonstration (proposition 4.1 page 120).

- Si $p_1 \sqsubseteq p_2 \sqsubseteq p_3$ alors $L(p_1) = L(p_2)_{|Dom(p_1)}$ et $L(p_2) = L(p_3)_{|Dom(p_2)}$ donc $L(p_1) = L(p_3)_{|Dom(p_2) \cap Dom(p_1)} = L(p_3)_{|Dom(p_1)}$ car $Dom(p_1) \subset Dom(p_2)$, donc $p_1 \sqsubseteq p_3$, donc $\sqsubseteq$ est transitif. Par ailleurs $p \sqsubseteq p$ car $L(p) = L(p)_{|Dom(p)}$ donc $\sqsubseteq$ est réflexif.
- $(\equiv) = (\sqsubseteq \cap \sqsupseteq)$ est la fermeture symétrique de $\sqsubseteq$.
- Si $p_1 \equiv_D p_2 \equiv_D p_3$ alors $L(p_1)_{|D} = L(p_2)_{|D} = L(p_3)_{|D}$, donc $p_1 \equiv_D p_3$, donc $\equiv_D$ est transitif.
- Si $q' \in Equiv_{\text{str}}(q)$, c.-à-d. $q' \equiv q$, alors $q \sqsubseteq q'$ donc $q' \in Equiv_{\text{par}}(q)$.
  Si $q' \in Equiv_{\text{par}}(q)$, alors $p \sqsubseteq q \sqsubseteq q'$ donc $q' \in Equiv_{\text{par}}(p)$.
- Si $p' \in Equiv_{\text{str}}(p)$ et $q' \in Equiv_{\text{str}}(q)$ et $q'' \in Equiv_{\text{par}}(q)$ alors $p' \equiv p \sqsubseteq q \equiv q' \equiv q \sqsubseteq q''$ donc $p' \sqsubseteq q' \sqsubseteq q''$.
- Si $q \in Equiv_{\text{par}}(p)$ alors $L(p) = L(q)_{|Dom(p)}$, donc $L(p)_{|D} = L(q)_{|D}$ car $D \subset Dom(p)$, donc $q \in Equiv_D(p)$.
  Si $q \in Equiv_D(p)$ alors $L(q)_D = L(p)_D$ donc $L(p)_{|D'} = L(q')_{|D'}$ car $D' \subset D$, donc $q \in Equiv_{D'}(p)$.
- Si $p_1 \in Equiv_{\text{str}}(p)$ et $p_2 \in Equiv_{\text{par}}(p)$ et $p_3 \in Equiv_D(p)$ alors $L(p_1)_{|D} = L(p_2)_{|D} = L(p_3)_{|D}$ car $Dom(p_1) \supset Dom(p_2) = Dom(p) \supset D \subset Dom(p_3)$ donc $p_1 \equiv_D p_2 \equiv_D p_3$.

Si L est une injection et $p' \in Equiv_{\text{str}}(p)$, alors $L(p') = L(p)$, donc $p' = p$. $\square$

Démonstration (proposition 4.2 page 120).

- Supposons $p \sqsubseteq p'$, c.-à-d. $(L\,p) = (L\,p')_{|Dom(p)}$.
  - $\forall q \in \mathcal{P}\ \ L\ (p \circ q) = (L\ p) \circ (L\ q) = (L\ p')_{|Dom(p)} \circ (L\ q) = (L\ p') \circ (L\ q)_{|L(q)^{-1}(Dom(p))} = (L\,p') \circ (L\,q)_{|Dom(p \circ q))} = L\ (p' \circ q)_{|Dom(p \circ q))}$, donc $p \circ q \sqsubseteq p' \circ q$.
  - $\forall q \in \mathcal{P}\ \ L\ (q \circ p) = (L\ q) \circ (L\ p) = (L\ q) \circ (L\ p')_{|Dom(p)} = (L\ q) \circ (L\ p')_{|L(p)^{-1}(Dom(q))} = L\ (q \circ p')_{|Dom(q \circ p))}$, donc $q \circ p \sqsubseteq q \circ p'$.
- Si $p \equiv p'$, c.-à-d. $p \sqsubseteq p'$ et $p \sqsupseteq p'$, alors $\forall q \in \mathcal{P}\ \ q \sqsubseteq p' \circ q$ et $q \circ p \sqsubseteq q \circ p'$ et $q \sqsupseteq p' \circ q$ et $q \circ p \sqsupseteq q \circ p'$ donc $p \circ q \equiv p' \circ q$ et $q \circ p \equiv q \circ p'$.
- Si $(L\,p)_{|D} = (L\,p')_{|D}$ alors $(L\,p)_{|D} \circ (Lq)_{|L(q)^{-1}(D)} = (L\,p')_{|D} \circ (Lq)_{|L(q)^{-1}(D)}$ car $L(q)(L(q)^{-1}(D)) \subset D$, donc $L\ (p \circ q)_{|L(q)^{-1}(D)} = L\ (p' \circ q)_{|L(q)^{-1}(D)}$. D'autre part $(L\ q) \circ (L\ p)_{|D} = (L\ q) \circ (L\ p')_{|D}$ donc $L\ (p \circ q)_{|D} = L\ (p' \circ q)_{|D}$. $\square$

Démonstration (proposition 4.3 page 121).

(1) • Si $x \in \mathrm{Emin}_{\preccurlyeq}(E) \cap A$, c.-à-d. $x \in A$ et $\forall y \in E\ \ y \preccurlyeq x \Rightarrow x \preccurlyeq y$, alors $x \in A$ et $\forall y \in A\ \ y \preccurlyeq x \Rightarrow x \preccurlyeq y$ car $A \subset E$, donc $x \in \mathrm{Emin}_{\preccurlyeq}(A)$.

- $x \in \mathrm{Emin}_{\preccurlyeq}(A)$ ssi $\forall y \in A\ \ y \preccurlyeq x \Rightarrow x \preccurlyeq y$ ssi $\forall y \in A\ \ y \not\preccurlyeq x \vee x \preccurlyeq y$ ssi $\forall y \in A\ \ \neg(y \preccurlyeq x \wedge x \not\preccurlyeq y)$ ssi $\forall y \in A\ \ x \not\succ y$ ssi $x \in \mathrm{Min}_{\not\succ}(A)$.

(2) • Si $x \in (\mathrm{Min}_{\preccurlyeq}(E) \cap A)$, c.-à-d. $x \in A$ et $\forall y \in E\ x \preccurlyeq y$, alors $x \in A$ et $\forall y \in A\ x \preccurlyeq y$ car $A \subset E$, donc $x \in \mathrm{Min}_{\preccurlyeq}(A)$.
• Si $x \in \mathrm{Min}_{\preccurlyeq}(A)$, c.-à-d. $x \in A$ et $\forall y \in A\ x \preccurlyeq y$ alors $\forall y \in A\ y \preccurlyeq x \Rightarrow x \preccurlyeq y$, donc $x \in \mathrm{Emin}_{\preccurlyeq}(A)$

(3) • $x \in \mathrm{Minor}_{A,\preccurlyeq}(A)$ ssi $x \in A$ et $\forall y \in A\ x \preccurlyeq y$ ssi $x \in \mathrm{Min}_{\preccurlyeq}(A)$.
• $x \in (A \cap \mathrm{Minor}_{E,\preccurlyeq}(A))$ ssi $x \in A$ et $x \in E$ et $\forall y \in A\ x \preccurlyeq y$ ssi $x \in \mathrm{Min}_{\preccurlyeq}(A)$.
• On a $\mathrm{Minor}_{A,\preccurlyeq}(A) = \mathrm{Min}_{\preccurlyeq}(A)$, donc $\mathrm{Minor}_{E,\preccurlyeq}(E) = \mathrm{Min}_{\preccurlyeq}(E)$ pour $A = E$, donc $\mathrm{Minor}_{E,\preccurlyeq}(E)_{|A} = \mathrm{Min}_{\preccurlyeq}(E)_{|A}$.

(4) • $x \in \mathrm{Inf}_{A,\preccurlyeq}(A)$ ssi $x \in A$ et $\forall y \in A\ x \preccurlyeq y$ et $\forall z \in A\ (\forall y \in A\ z \preccurlyeq y) \Rightarrow z \preccurlyeq x$, or si $\forall y \in A\ z \preccurlyeq y$ alors $z \preccurlyeq x$ car $x \in A$ donc $x \in \mathrm{Inf}_{A,\preccurlyeq}(A)$ ssi $x \in A$ et $\forall y \in A\ x \preccurlyeq y$ ssi $x \in \mathrm{Min}_{\preccurlyeq}(A)$.
• $x \in \mathrm{Inf}_{E,\preccurlyeq}(A)_{|A}$ ssi $x \in A$ et $\forall y \in A\ x \preccurlyeq y$ et $\forall z \in E\ (\forall y \in A\ z \preccurlyeq y) \Rightarrow z \preccurlyeq x$, or si $\forall y \in A\ z \preccurlyeq y$ alors $z \preccurlyeq x$ car $x \in A$ donc $x \in \mathrm{Inf}_{E,\preccurlyeq}(A)$ ssi $x \in A$ et $\forall y \in A\ x \preccurlyeq y$ ssi $x \in \mathrm{Min}_{\preccurlyeq}(A)$.
• On a $\mathrm{Inf}_{A,\preccurlyeq}(A) = \mathrm{Min}_{\preccurlyeq}(A)$, donc $\mathrm{Inf}_{E,\preccurlyeq}(E) = \mathrm{Min}_{\preccurlyeq}(E)$ pour $A = E$, donc $\mathrm{Inf}_{E,\preccurlyeq}(E)_{|A} = \mathrm{Min}_{\preccurlyeq}(E)_{|A}$.

(5) Soient $x_1 \in (\mathrm{Emin}_{\preccurlyeq}(E) \cap A)$, $x_2 \in (\mathrm{Min}_{\preccurlyeq}(E) \cap A)$, $x_3 \in \mathrm{Min}_{\preccurlyeq}(A)$, $x_4 \in \mathrm{Emin}_{\preccurlyeq}(A)$, $x_5 \in A$. Alors $\forall y \in E\ y \preccurlyeq x_1 \Rightarrow x_1 \preccurlyeq y$ et $\forall y \in E\ x_2 \preccurlyeq y$ et $\forall y \in A\ x_3 \preccurlyeq y$ et $\forall y \in A\ y \preccurlyeq x_4 \Rightarrow x_4 \preccurlyeq y$, donc $x_2 \preccurlyeq x_1, x_3, x_4, x_5$ et $x_3 \preccurlyeq x_1, x_2, x_4, x_5$ donc $x_1 \preccurlyeq x_2$ et $x_4 \preccurlyeq x_3$. Par conséquent $x_1 \approx x_2 \approx x_3 \approx x_4$ et $x_2, x_3 \preccurlyeq x_5$.

La relation $\preccurlyeq$ est totale ssi $\preccurlyeq \cup \succcurlyeq\ = \ \approx_{\mathrm{triv}}$. On a dans ce cas $\not\succ\ = \complement(\not\preccurlyeq \cap \succcurlyeq) = \preccurlyeq \cup \not\succcurlyeq\ = (\preccurlyeq \cup \not\succcurlyeq) \cap \approx_{\mathrm{triv}} = (\preccurlyeq \cup \not\succcurlyeq) \cap (\preccurlyeq \cup \succcurlyeq) = \preccurlyeq \cap (\not\succcurlyeq \cup \succcurlyeq) = \preccurlyeq \cap \approx_{\mathrm{triv}} = \preccurlyeq$. Par conséquent $\mathrm{Emin}_{\preccurlyeq}(A) = \mathrm{Min}_{\not\succ}(A) = \mathrm{Min}_{\preccurlyeq}(A)$. ☐

Démonstration (tableau 4.1 page 122).

Plus petits éléments.

- Relation stricte : $x \in \mathrm{Min}_{\prec}(A)$ ssi $x \in A$ et $\forall y \in A\ x \prec y$. Or $x \not\prec x$, donc $\mathrm{Min}_{\prec}(A) = \varnothing$.
- Inclusion : déduit de l'intersection.
- Intersection : $x \in \mathrm{Min}_{\cap_{i \in I} \preccurlyeq_i}(A)$ ssi $x \in A$ et $\forall y \in A\ \forall i \in I\ x \preccurlyeq_i y$ ssi $\forall i \in I\ (x \in A$ et $\forall y \in A\ x \preccurlyeq_i y)$ ssi $x \in \bigcap_{i \in I} \mathrm{Min}_{\preccurlyeq_i}(A)$.
- Réunion : $x \in \mathrm{Min}_{\cup_{i \in I} \preccurlyeq_i}(A)$ ssi $x \in A$ et $\forall y \in A\ \exists i \in I\ x \preccurlyeq_i y$ si $\exists i \in I\ (x \in A$ et $\forall y \in A\ x \preccurlyeq_i y)$ ssi $x \in \bigcup_{i \in I} \mathrm{Min}_{\preccurlyeq_i}(A)$.
- « $A \subset A'$ » : On a vu (prop. 4.3.2) que $\mathrm{Min}_{\preccurlyeq}(A') \cap A = \mathrm{Min}_{\preccurlyeq|A}(A') \subset \mathrm{Min}_{\preccurlyeq}(A)$, donc $\mathrm{Min}_{\preccurlyeq}(A) \supset \mathrm{Min}_{\preccurlyeq}(A')$ et $\mathrm{Min}_{\preccurlyeq|A}(E) = \mathrm{Min}_{\preccurlyeq}(E) \cap A \subset \mathrm{Min}_{\preccurlyeq}(E) \cap A' = \mathrm{Min}_{\preccurlyeq|A'}(E)$.

Pour la restriction à A, on a exactement les mêmes propriétés car « $\cap A$ » se distribue sur l'intersection et la réunion.

Éléments minimaux.

- Relation stricte : Si $f(\preccurlyeq) = \not\succ\ = \complement(\not\preccurlyeq \cap \succcurlyeq) = \preccurlyeq \cup \not\succcurlyeq$ alors $f(\prec) = (\preccurlyeq \cap \not\succcurlyeq) \cup \complement(\not\preccurlyeq \cap \succcurlyeq) = (\preccurlyeq \cap \not\succcurlyeq) \cup (\preccurlyeq \cup \not\succcurlyeq) = \preccurlyeq \cup \not\succcurlyeq\ = f(\preccurlyeq)$. Or $\mathrm{Emin}_{\preccurlyeq}(A) = \mathrm{Min}_{\not\succ}(A)$ donc $\mathrm{Emin}_{\prec}(A) = \mathrm{Min}_{\not\succ}(A) = \mathrm{Emin}_{\preccurlyeq}(A)$.
- Intersection : Soit $\preccurlyeq\ = \bigcap_{i \in I} \preccurlyeq_i$. On a $\mathrm{Emin}_{\preccurlyeq}(A) = \mathrm{Min}_{\not\succ}(A)$. Or $\not\succ\ \supset \bigcap_{i \in I} \not\succ_i$ (tabl. 3.1), donc $\mathrm{Min}_{\not\succ}(A) \supset \mathrm{Min}_{\cap_{i \in I} \not\succ_i}(A) \supset \bigcap_{i \in I} \mathrm{Min}_{\not\succ_i}(A) = \bigcap_{i \in I} \mathrm{Emin}_{\preccurlyeq_i}(A)$.
- « $A \subset A'$ » : l'argument est identique au cas Min car $\mathrm{Emin}_{\preccurlyeq}(A') \cap A = \mathrm{Emin}_{\preccurlyeq|A}(A') \subset \mathrm{Emin}_{\preccurlyeq}(A)$ (prop. 4.3.1).

Pour la restriction à A, on a aussi les mêmes propriétés car « $\cap A$ » se distribue sur l'intersection et la réunion. □

Démonstration (proposition 4.4 page 123).

(1) $p \preccurlyeq p$ et $p \in Equiv_{\mathrm{str}}(p) \subset P$. C'est l'analogue du programme résiduel trivial (cf. §1.3.1).

(2) $q \in Evap^{\mathrm{optF}}(p)$ ssi $q \preccurlyeq p$ et $\forall q' \in P \;\; q \preccurlyeq q'$ ssi $\forall q' \in P \;\; q \preccurlyeq q'$ ssi $\forall q' \in P \;\; q \in Evap(q')$ ssi $q \in \bigcap_{q' \in P} Evap(q')$.

(3) $q \in Evap^{\mathrm{optf}}(p)$ ssi $q \preccurlyeq p$ et $\forall q' \in P \;\; q' \preccurlyeq p \;\Rightarrow\; q \preccurlyeq q'$ ssi $q \in Evap(p)$ et $\forall q' \in Evap(p)$ $q \in Evap(q')$ ssi $q \in Evap(p) \cap (\bigcap_{q' \in \mathrm{Evap}(p)} Evap(q'))$. De plus si $Evap(p) \neq \varnothing$, c.-à-d. $\exists q_0 \in P \;\; q_0 \preccurlyeq p$ alors $q \preccurlyeq q_0 \preccurlyeq p$. La condition $q \in Evap(p)$ est donc superflue.

(4) $q \in Evap^{\mathrm{minF}}_{P,\preccurlyeq}(p)$ ssi $q \preccurlyeq p$ et $\forall q' \in P \;\; q' \not\prec q$ ssi $q \in Evap_{P,\preccurlyeq}(p)$ et $\forall q' \in P \;\; q \in Evap_{P,\not\succ}(q')$ ssi $q \in Evap_{P,\preccurlyeq}(p) \cap (\bigcap_{q' \in P} Evap_{P,\not\succ}(q'))$.

(5) $q \in Evap^{\mathrm{minF}}_{P,\preccurlyeq}(p)$ ssi $q \preccurlyeq p$ et $\forall q' \in P \;\; q' \preccurlyeq p \;\Rightarrow\; q \not\succ q'$ ssi $q \in Evap_{P,\preccurlyeq}(p)$ et $\forall q' \in Evap_{P,\preccurlyeq}(p)$ $q \in Evap_{P,\not\succ}(q')$ ssi $q \in Evap_{P,\preccurlyeq}(p) \cap (\bigcap_{q' \in \mathrm{Evap}_{P,\preccurlyeq}(p)} Evap_{P,\not\succ}(q'))$.

(6) Soit $q \preccurlyeq p$. On a $q \notin Evap^{\mathrm{minF}}(p)$ ssi $\exists q' \in P \;\; q' \preccurlyeq q \;\wedge\; q \not\preccurlyeq q'$, donc $q' \preccurlyeq p$ par transitivité, donc $\exists q' \in P \;\; q' \preccurlyeq p \;\wedge\; q' \preccurlyeq q \;\wedge\; q \not\preccurlyeq q'$, donc $q \notin Evap^{\mathrm{minf}}(p)$. Par conséquent $Evap^{\mathrm{minf}}(p) \subset Evap^{\mathrm{minF}}(p)$. Le reste des inclusions est donné par les points (1) et (2) de la proposition 4.3 avec $E = P$ et $A = Evap_{P,\preccurlyeq}(p)$.
Si $\preccurlyeq$ est totale, Emin est équivalent à Min (prop. 4.3.2), donc $Evap^{\mathrm{optF}} = Evap^{\mathrm{minF}}$.

(7) C'est la traduction du point (5) de la proposition 4.3. Parce que $\approx$ est transitive, l'ordre des ensembles dans cette formule est sans importance.

(8) On sait que $Equiv_{\mathrm{str}}(p) = \{p\}$ (prop. 4.1). On a trivialement $p \in Evap^{\mathrm{optF}}_{\mathrm{str}}(p)$; les inclusions du point (6) sont donc saturées.

La section §4.4 recelle quelques exemples d'inclusions strictes. □

Démonstration (tableau 4.2 page 124). Nous donnons uniquement les propriétés de l'évaluation partielle simple. Celles des évaluations partielles optimales et minimales sont déduites du tableau 4.1.

- Relation stricte : $Evap_{\prec}(p) = Evap_{\preccurlyeq \cap \not\succcurlyeq}(p) = Evap_{\preccurlyeq}(p) \cap (Evap_{\not\succcurlyeq}(p)) \subset Evap_{\preccurlyeq}(p)$.
- Inclusion : elle est déduite de l'intersection.
- Intersection : $q \in Evap_{\cap_{i \in I} \preccurlyeq_i}(p)$ ssi $\forall i \in I \;\; q \preccurlyeq_i p$ ssi $q \in (\bigcap_{i \in I} Evap_{\preccurlyeq_i}(p))$.
- Réunion : $q \in Evap_{\cup_{i \in I} \preccurlyeq_i}(p)$ ssi $\exists i \in I \;\; q \preccurlyeq_i p$ ssi $q \in (\bigcup_{i \in I} Evap_{\preccurlyeq_i}(p))$.
- « $P \subset P'$ » : $q \in Evap_P(p)$ ssi $q \in P$ et $q \preccurlyeq p$ ssi $q \in P'$ et $q \preccurlyeq p$ et $q \in P$ ssi $q \in (Evap_{P'}(p) \cap P)$.

Soit P un ensemble qui vérifie $Equiv_{\mathrm{str}}(p) = Equiv_{\mathrm{str}}(p') \subset P \subset Equiv_{\mathrm{par}}(p) = Equiv_{\mathrm{par}}(p')$.

- $Evap$: si $q \in Evap_P(p)$ alors $q \in P$ et $q \preccurlyeq p \preccurlyeq p'$ donc $q \in Evap_P(p')$, et donc $Evap_P(p) \subset Evap_P(p')$.
- $Evap^{\min}$: si $q \in Evap^{\min}_P(p)$ alors $q \in P$ et $q \preccurlyeq p \preccurlyeq p'$ et $\forall q' \in P \;\; q' \preccurlyeq q \;\Rightarrow\; q \preccurlyeq q'$ donc $q \in Evap^{\min}_P(p')$, et donc $Evap^{\min}_P(p) \subset Evap^{\min}_P(p')$.
- $Evap^{\mathrm{optf}}$: si $q \in Evap^{\mathrm{optf}}_P(p')$, alors $q \in P$ et $q \preccurlyeq p'$ et $\forall q' \in P \;\; q' \preccurlyeq p' \;\Rightarrow\; q \preccurlyeq q'$, or $p \in P$ et $p \preccurlyeq p'$, donc $q \preccurlyeq p$. De plus $\forall q' \in P$ si $q' \preccurlyeq p$ alors $q' \preccurlyeq p'$, donc $q \preccurlyeq q'$, donc $q \in Evap^{\mathrm{optf}}_P(p')$, et donc $Evap^{\mathrm{optf}}_P(p) \subset Evap^{\mathrm{optf}}_P(p')$.
- $Evap^{\mathrm{optF}}$: l'évaluation partielle optimale forte ne dépend que de P (prop. 4.4.2), donc $Evap^{\mathrm{optF}}_P(p) = Evap^{\mathrm{optF}}_P(p')$. □

Démonstration (proposition 4.6 page 139).

- Soit $x \in \mathrm{Emin}_{\preccurlyeq}(A)$, c'est-à-dire $x \in A$ et pour tout $y \in A$, si $y \preccurlyeq x$ alors $x \preccurlyeq y$. Pour tout $y' \in \mathrm{Emin}_{\preccurlyeq'}(f'(A))$, si $y' \preccurlyeq' f'(x)$ alors $f(y') \preccurlyeq f(f'(x)) \preccurlyeq x$ donc $x \preccurlyeq f(y')$ par hypothèse sur x. Par conséquent, $f'(x) \preccurlyeq' f'(f(y')) \preccurlyeq' y'$. Autrement dit, $f'(x) \in \mathrm{Emin}_{\preccurlyeq'}(f'(A))$.
- $f'(\mathrm{Emin}_{\preccurlyeq|A}(E)) = f'(\mathrm{Emin}_{\preccurlyeq}(E) \cap A) \subset f'(\mathrm{Emin}_{\preccurlyeq}(E)) \cap f'(A) \subset \mathrm{Emin}_{\preccurlyeq'}(f'(E)) \cap f'(A) = \mathrm{Emin}_{\preccurlyeq'|f'(A)}(f'(E))$
- Soit $x \in \mathrm{Min}_{\preccurlyeq}(A)$, c'est-à-dire pour tout $y \in A$, $x \preccurlyeq y$. Pour tout $y' \in \mathrm{Min}_{\preccurlyeq'}(f'(A))$, on a en particulier $x \preccurlyeq f(y')$ et donc $f'(x) \preccurlyeq' f'(f(y')) \preccurlyeq' y'$. Autrement dit, $f'(x) \in \mathrm{Min}_{\preccurlyeq'}(f'(A))$.
- $f'(\mathrm{Min}_{\preccurlyeq|A}(E)) = f'(\mathrm{Min}_{\preccurlyeq}(E) \cap A) \subset \mathrm{Min}_{\preccurlyeq'}(f'(E)) \cap f'(A) = \mathrm{Min}_{\preccurlyeq'|f'(A)}(f'(E))$ □

Démonstration (proposition 4.7 page 140). Si $P = \mathit{Equiv}_{\mathrm{str}}(p)$ et $P' = \mathit{Equiv}_{\mathrm{str}}(T'(p))$, alors pour tout $q \in P$, $T'(q) \equiv T'(p)$ donc, $T'(P) \subset P'$. On a de même $T(P') \subset P$. La situation est identique pour l'extension paresseuse : si $P = \mathit{Equiv}_{\mathrm{par}}(p)$ et $P' = \mathit{Equiv}_{\mathrm{par}}(T'(p))$, alors pour tout $q \in P$, $T'(q) \sqsupseteq T'(p)$, donc $T'(P) \subset P'$. De même, pour tout $q' \in \mathcal{P}'$, $T(q') \sqsupseteq T(T'(p)) \sqsupseteq p$, donc $T(P') \subset P$. Ce sont les seules hypothèses que nous utiliseront sur P et P'. Les démonstrations qui suivent utilisent principalement les lignes 5 et 7 du tableau 4.2.

- Soit $q \in \mathit{Evap}_{P,}(p)$, c'est-à-dire $q \in P$ et $q \preccurlyeq p$. Alors $T'(q) \in T'(P)$ et $T'(q) \preccurlyeq' T'(p)$. Autrement dit, $T'(q) \in \mathit{Evap}_{T'(P)}(T'(p)) \subset \mathit{Evap}_{P'}(T'(p))$ car $T'(P) \subset P'$ (tabl. 4.2).
- $\mathit{Evap}_P^{\mathrm{optF}}(p) = \mathrm{Min}_{|\mathit{Evap}_P(p)}(P)$ par définition, donc $T'(\mathit{Evap}_P^{\mathrm{optF}}(p)) \subset \mathrm{Min}_{|T'(\mathit{Evap}_P(p))}(T'(P))$ d'après la proposition 4.6. Or $T'(\mathit{Evap}_P(p)) \subset \mathit{Evap}_{P'}(T'(p))$ d'après le point précédent. On a donc bien l'inclusion dans $\mathrm{Min}_{|\mathit{Evap}_{P'}(T'(p))}(P')$ (tabl. 4.2). Et cet ensemble n'est autre que $\mathit{Evap}_{P'}^{\mathrm{optF}}(T'(p))$.
- D'après le point (3) de la proposition 4.4, on a $\mathit{Evap}_P^{\mathrm{optf}}(p) = \mathit{Evap}_P(p) \cap (\bigcap_{q \in \mathit{Evap}_P(p)} \mathit{Evap}_P(q))$. Par conséquent $T'(\mathit{Evap}_P^{\mathrm{optf}}(p)) \subset T'(\mathit{Evap}_P(p)) \cap (\bigcap_{q \in \mathit{Evap}_P(p)} T'(\mathit{Evap}_P(q))) \subset \mathit{Evap}_{P'}(T'(p)) \cap (\bigcap_{q \in \mathit{Evap}_P(p)} \mathit{Evap}_{P'}(T'(q)))$.
  Pour montrer ensuite que $\bigcap_{q \in \mathit{Evap}_P(p)} \mathit{Evap}_{P'}(T'(q)) \subset \bigcap_{q' \in \mathit{Evap}_{P'}(T'(p))} \mathit{Evap}_{P'}(q')$, on vérifie que chaque terme présent dans la seconde intersection contient un terme présent dans la première. Soit donc $q' \in \mathit{Evap}_{P'}(T'(p))$. Alors le programme $q = T(q')$ appartient à $\mathit{Evap}_{T(P')}(T(T'(p))) \subset \mathit{Evap}_P(p)$ car $T(P') \subset P$ et $T(T'(p)) \preccurlyeq p$ (tabl. 4.2), et il vérifie $\mathit{Evap}_{P'}(T'(q)) = \mathit{Evap}_{P'}(T'(T(q'))) \subset \mathit{Evap}_{P'}(q')$). En résumé, on a $T'(\mathit{Evap}_P^{\mathrm{optf}}(p)) \subset \mathit{Evap}_{P'}(T'(p)) \cap (\bigcap_{q' \in \mathit{Evap}_{P'}(T'(p))} \mathit{Evap}_{P'}(q')) = \mathit{Evap}_{P'}^{\mathrm{optf}}(T'(p)))$.
- $\mathit{Evap}_P^{\min}(p) = \mathrm{Emin}(\mathit{Evap}_P(p))$ par définition, donc $T'(\mathit{Evap}_P^{\min}(p)) \subset \mathrm{Emin}(T'(\mathit{Evap}_P(p)))$ d'après la proposition 4.6. Or $T'(\mathit{Evap}_P(p)) \subset \mathit{Evap}_{P'}(T'(p))$. Par conséquent, on a donc bien l'inclusion dans $\mathrm{Emin}(\mathit{Evap}_{P'}(T'(p))) = \mathit{Evap}_{P'}^{\min}(T'(p))$.
- $T(T'(\mathit{Evap}_P(p))) \subset T(\mathit{Evap}_{P'}(T'(p))) \subset \mathit{Evap}_P(T(T'(p)))) \subset \mathit{Evap}_P(p)$ car $T(T'(p)) \preccurlyeq p$ (tabl. 4.2). Le raisonnement est identique pour les évaluations partielles optimale forte et minimale. Il y a de plus égalité pour l'évaluation partielle optimale forte car elle est indépendante de p.
- En ce qui concerne l'évaluation partielle optimale faible, nous avons toujours $\mathit{Evap}_P^{\mathrm{optf}}(T(T'(p))) = \mathit{Evap}_P(T(T'(p))) \cap (\bigcap_{q \in \mathit{Evap}_P(T(T'(p)))} \mathit{Evap}_P(q))$. Comme nous l'avons déja fait ci-dessus, pour montrer l'inclusion de $\bigcap_{q \in \mathit{Evap}_P(T(T'(p)))} \mathit{Evap}_P(q)$ dans $\bigcap_{q \in \mathit{Evap}_P(p)} \mathit{Evap}_P(q)$, nous vérifions que chaque terme présent dans la seconde intersection contient un terme présent dans la première. Soit donc $q \in \mathit{Evap}_P(p)$. Alors le programme $T(T'(q))$ appartient à $\mathit{Evap}_P(T(T'(p))) \subset \mathit{Evap}_P(p)$ car $T(T'(p)) \preccurlyeq p$ (tabl. 4.2), et il vérifie $\mathit{Evap}_P(T(T'(q))) \subset \mathit{Evap}_P(q)$. Par conséquent, $\mathit{Evap}_P^{\mathrm{optf}}(T(T'(p))) \subset \mathit{Evap}_P(p) \cap (\bigcap_{q \in \mathit{Evap}_P(p)} \mathit{Evap}_P(q)) = \mathit{Evap}_P^{\mathrm{optf}}(p)$.

Notons que pour l'évaluation partielle optimale faible, il n'est pas possible d'utiliser un argument similaire à l'évaluation partielle optimale forte ou minimale car elle ne vérifie pas la propriété donnée à la 5ème ligne du tableau 4.2. C'est pourquoi il nous a fallu détailler davantage. □

Démonstration (proposition 4.8 page 141). Pour tous programmes p et q de $\mathcal{P}$, on a $p \equiv q$ ssi $L(p) = L(q)$ ssi $L'(T'(p)) = L'(T'(q))$ ssi $T'(p) \equiv T'(q)$. Le résultat est identique pour T. Par ailleurs, $L(p) = L'(T'(p)) = L(T(T'(p)))$ donc $p \equiv T(T'(p))$. On a donc bien une connexion de Galois (contractante) entre $(\mathcal{P}, \equiv)$ et $(\mathcal{P}', \equiv')$.

D'autre part, pour tout $p \in \mathcal{P}$ et tous $q_1, q_2 \in \mathit{Equiv}_{\mathrm{str}}(p)$, on a $L(q_1) = L(p) = L(q_2)$, c.-à-d. $L'(T'(q_1)) = L'(T'(q_2))$. Puisque L' est injective, on a $T'(q_1) = T'(q_2)$. On a alors trivialement $\mu'(T'(q_1)) = \mu'(T'(q_2))$, et donc $T'(q_1) \approx_{\mu'} T'(q_2)$ car $\preccurlyeq$ est réflexif. Réciproquement, si $q_1', q_2' \in \mathit{Equiv}_{\mathrm{str}}(p')$ pour $p' \in \mathcal{P}'$, on a $L'(q_1') = L'(p') = L'(q_2')$, donc $q_1' = q_2'$ et $T(q_1) \approx_{\mu} T(q_2)$. Enfin, pour tout programmes p de $\mathcal{P}$, on a $\mu(T(T'(p))) \preccurlyeq \mu'(T'(p)) \preccurlyeq \mu(p)$, c'est-à-dire $T(T'(p)) \preccurlyeq_{\mu} p$. On a symétriquement le même résultat pour $\mathcal{P}'$. Par conséquent, (T', T) forme bien une correspondance stricte entre L et L'.

Pour conclure, nous savons que pour tout $p' \in \mathcal{P}'$, on a $p' \in \mathit{Evap}^{\mathrm{optF}}_{\mathrm{str}}(p) = \{p'\}$ (prop. 4.4.8). En particulier, pour tout $p \in \mathcal{P}$, on a $T'(p) \in \mathit{Evap}^{\mathrm{optF}}(T'(p))$, donc $T(T'(p)) \in \mathit{Evap}^{\mathrm{optF}}(p)$ (prop. 4.7). □

D.5 Transformations et correction.

Démonstration (lemme 5.1 page 159). Soit une dérivation $M \leftarrow_{u_0, r_0, \sigma_0} N \twoheadrightarrow_{(u, r_u, \sigma_u)_{u \in U}} P$ avec $r_0 = \langle \alpha_0 \to \beta_0 \rangle$ et $r_u = \langle \alpha_u \to \beta_u \rangle$ pour $u \in U$. Par définition, cela signifie $N/u_0 = \sigma_0 \alpha_0$, $M = N[u_0 \leftarrow \sigma_0 \beta_0]$, et pour tout $u \in U$, $N/u = \sigma_u \alpha_u$, et $P = N[u \leftarrow \sigma_u \beta_u]$.

Par hypothèse, pour tout $u \in U$, r_u ne se superpose pas à r_0 en u/u_0. Autrement dit, $u/u_0 \notin \mathit{Occ}(\alpha_0)$ ou $\alpha_0(u/u_0) \in \mathit{Var}(\alpha_0)$ ou $\{\alpha_0/(u/u_0), \alpha_u\}$ n'est pas unifiable. Or $u_0 \leqslant u$ et $\sigma_u \alpha_u = N/u = (M[u_0 \leftarrow \sigma_0 \alpha_0])/u = \sigma_0 \alpha_0/(u_0/u)$ donc $\{\alpha_0/(u/u_0), \alpha_u\}$ est unifiable. Il ne subsiste que $u/u_0 \notin \mathit{Occ}(\alpha_0)$ ou $\alpha_0(u/u_0) \in \mathit{Var}(\alpha_0)$.

Dans chaque cas, puisque $u_0 \leqslant u$, l'occurrence u/u_0 pointe sous une variable x_u de $\mathit{Var}(\alpha_0)$ dans le terme $N/u_0 = \sigma_0 \alpha_0$. Autrement dit, il existe une occurrence $v_u \in \mathit{Occ}_{x_u}(\alpha_0)$ telle que $v_u \leqslant u/u_0$. Soit $w_u = (u/u_0)/v_u = u/(u_0 \cdot v_u)$; c'est l'occurrence de la dérivation par r_u dans $\sigma_0 x_u$. Nous avons $\sigma_0 x_u / w_u = N/u = \sigma_u \alpha_u$. Notons $X_U = \{x_u \mid u \in U\}$ l'ensemble des variables de α_0 traversées par U et, pour $x \in X_U$, $U_x = \{u \in U \mid x_u = x\}$ l'ensemble des occurrences de U qui traversent dans N une variable x donnée.

1. On cherche à relier M à P en utilisant les règles $(r_u)_{u \in U}$ avant r_0. Pour cela, on suit les traces des occurrences de U à travers la dérivation ∂_0. Ces traces ont la propriété d'être parallèles entre elles. En effet, les familles suivantes sont constituées d'occurrences disjointes :
   - $(u)_{u \in U}$ par définition
   - $(u/u_0)_{u \in U} = (v_u \cdot w_u)_{u \in U}$ car $u_0 \leqslant U$
   - $(v_u \cdot w_u)_{u \in U, v_u = v}$ pour $x \in X_U$ et $v \in \mathit{Occ}_x(\alpha_0)$ en tant que sous-famille
   - $(w_u)_{u \in U, v_u = v}$ pour $x \in X_U$ et $v \in \mathit{Occ}_x(\alpha_0) = \{v\}$ car α_0 est linéaire
   - $(v' \cdot w_u)_{u \in U, v_u = v}$ pour $x \in X_U$ et $v \in \mathit{Occ}_x(\alpha_0)$ et $v' \in \mathit{Occ}_x(\beta_0)$
   - $(u_0 \cdot v' \cdot w_u)_{u \in U, v' \in \mathit{Occ}_{x_u}(\beta_0)}$ car $\mathit{Occ}_X(\beta_0)$ est constitué d'occurrences disjointes

   L'ensemble $U' = \bigcup_{u \in U} u_0 \cdot \mathit{Occ}_{x_u}(\beta_0) \cdot w_u$ est la trace des occurrences de U dans le terme M. On peut donc définir une dérivation parallèle $\partial' = (u', r'_{u'}, \sigma'_{u'})_{u' \in U'}$ par $r'_{u'} = r_u$ et $\sigma'_{u'} = \sigma_u$ où u est l'unique occurrence de U telle que $u' \in u_0 \cdot \mathit{Occ}_{x_u}(\beta_0) \cdot w_u$.

Pour tout $v' \in Occ_{x_u}(\beta_0)$, on a $M/u_0 \cdot v' \cdot w_u = \sigma_0\beta_0/v' \cdot w_u = \sigma_0 x_u/w_u = \sigma_u\alpha_u$. On a donc $M \twoheadrightarrow_{\partial'} M'$ avec $M' = M[u_0 \cdot Occ_{x_u}(\beta_0) \cdot w_u \leftarrow \sigma_u\beta_u \mid u \in U]$. Notons que si β_0 est linéaire, alors U' contient au plus $|U|$ occurrences ; si β_0 est d'ordre zéro, alors U' est vide et $M' = M$.

2. Soit $x \in X_U$ et $v' \in Occ_x(\beta_0)$, alors le terme $M'/u_0 \cdot v' = M[u_0 \cdot Occ_{x_u}(\beta_0) \cdot w_u \leftarrow \sigma_u\beta_u \mid u \in U]/u_0 \cdot v' = (M/u_0 \cdot v')[w_u \leftarrow \sigma_u\beta_u \mid u \in U, x = x_u] = \sigma_0 x[w_u \leftarrow \sigma_u\beta_u \mid u \in U_x]$ ne dépend que de x et non de v' ; nous le notons t_x.

   Soit θ la substitution sur les variables de α_0 et β_0 définie par $\theta(x) = t_x$ si $x \in X_U$, et $\theta(x) = \sigma_0(x)$ si $x \notin (Var(\alpha_0) \cup Var(\beta_0)) \setminus X_U$.

   Alors $P/u_0 = N[u \leftarrow \sigma_u\beta_u \mid u \in U]/u_0 = (N/u_0)[u/u_0 \leftarrow \sigma_u\beta_u \mid u \in U] = \sigma_0\alpha_0[v_u \cdot w_u \leftarrow \sigma_u\beta_u \mid u \in U] = \sigma_0\alpha_0[Occ_{x_u}(\alpha_0) \cdot w_u \leftarrow \sigma_u\beta_u \mid u \in U] = \sigma_0\alpha_0[Occ_x(\alpha_0) \leftarrow \sigma_0 x[w_u \leftarrow \sigma_u\beta_u | u \in U_x] \mid x \in X_U] = \sigma_0\alpha_0[Occ_x(\alpha_0) \leftarrow t_x \mid x \in X_U] = \theta\alpha_0$.

   D'autre part, $M'/u_0 = (M/u_0)[Occ_{x_u}(\beta_0) \cdot w_u \leftarrow \sigma_u\beta_u \mid u \in U] = \sigma_0\beta_0[Occ_{x_u}(\beta_0) \cdot w_u \leftarrow \sigma_u\beta_u \mid u \in U] = \sigma_0\beta_0[Occ_x(\beta_0) \leftarrow \sigma_0 x[w_u \leftarrow \sigma_u\beta_u | u \in U_x] \mid x \in X_U] = \sigma_0\beta_0[Occ_x(\beta_0) \leftarrow t_x \mid x \in X_U] = \theta\beta_0$. Par conséquent, $M' \leftarrow_{\partial_0,\theta_0} P$.

En conclusion, on a bien $M \twoheadrightarrow_{\partial'} M' \leftarrow_{\partial_0} P$ avec $u_0 \leqslant U'$. $\square$

Démonstration (lemme 5.2 page 159). On montre $\forall\partial_1 \in Deriv(R_1)\ \forall\partial_2 \in Deriv(R_2)\ \exists\partial'_1 \in Deriv(R_1)\ \exists\partial'_2 \in Deriv(R_2)\ \twoheadleftarrow_{\partial_1}\twoheadrightarrow_{\partial_2} \subset \twoheadrightarrow_{\partial'_2}\twoheadleftarrow_{\partial'_1}$ avec $Roots(Occ(\partial_1) \cup Occ(\partial_2)) = Roots(Occ(\partial'_1) \cup Occ(\partial'_2))$ par récurrence sur $|\partial_1| + |\partial_2|$.

La proposition est triviale dans le cas $\partial_1 = \varnothing$ ou $\partial_2 = \varnothing$. Supposons $\partial_1 \neq \varnothing$ et $\partial_2 \neq \varnothing$. Soient $u_1 \in U_1 = Occ(\partial_1)$ et $U_2 = Occ(\partial_2)$. Soit r_1 la règle associée à u_1 dans ∂_1. La dérivation ∂_1 s'écrit $\partial_1 = (u_1, r_1) \cup \partial_{1|U_1\setminus\{u_1\}}$. Quatre positions relatives des occurrences sont possibles.

1. Cas $\exists u \in U_2\ \ u = u_1$. Soit alors $r_2 = \langle\alpha_2 \to \beta_2\rangle$ la régle associée à u dans ∂_2.
   - 1.1. Si r_2 ne se superpose pas à r_1 en ϵ, il existe par application du lemme 5.1 une dérivation parallèle ∂'_2 de r_2 dominée par u telle que $\leftarrow_{u,r_1}\rightarrow_{u,r_2} \subset \twoheadrightarrow_{\partial'_2}\leftarrow_{u,r_1}$.
   - 1.2. En revanche, si r_2 se superpose à r_1 en ϵ, c'est nécessairement une superposition impropre car $Crit_{\text{pro}}(R_1, R_2) = \varnothing$. Donc $\alpha_2 \in X$, et par conséquent r_1 ne se superpose pas à r_2 en ϵ. Par application du lemme 5.1, il existe donc une dérivation parallèle ∂'_1 de r_1 dominée par u telle que $\leftarrow_{u,r_1}\rightarrow_{u,r_2} \subset \rightarrow_{u,r_2}\twoheadleftarrow_{\partial'_1}$.

   En tout état de cause, il existe des dérivations parallèles ∂'_1 et ∂'_2 de r_1 et r_2, dominées par u, et telles que $\leftarrow_{u,r_1}\rightarrow_{u,r_2} \subset \twoheadrightarrow_{\partial'_2}\twoheadleftarrow_{\partial'_1}$ avec $Roots(\partial'_1, \partial'_2) = \{u\}$.

   Or $U_1 \setminus \{u\} \parallel \{u\}$ et $U_2 \setminus u \parallel \{u\}$, donc $\twoheadleftarrow_{\partial_1}\twoheadrightarrow_{\partial_2} = \twoheadleftarrow_{\partial_1|U_1\setminus\{u\}}\leftarrow_{u,r_1}\rightarrow_{u,r_2}\twoheadrightarrow_{\partial_2|U_2\setminus\{u\}} \subset \twoheadleftarrow_{\partial_1|U_1\setminus\{u\}}\twoheadrightarrow_{\partial'_2}\twoheadleftarrow_{\partial'_1}\twoheadrightarrow_{\partial_2|U_2\setminus\{u\}} = \twoheadrightarrow_{\partial'_2}\twoheadleftarrow_{\partial_1|U_1\setminus\{u\}}\twoheadrightarrow_{\partial_2|U_2\setminus\{u\}}\twoheadleftarrow_{\partial'_1}$.

   On peut appliquer l'hypothèse de récurrence à $\partial_{1|U_1\setminus\{u\}}$ et $\partial_{2|U_2\setminus\{u\}}$. Il existe donc des dérivations $\partial''_1 \in Deriv(R_1)$ et $\partial''_2 \in Deriv(R_2)$ telles que $\twoheadleftarrow_{\partial_1|U_1\setminus\{u\}}\twoheadrightarrow_{\partial_2|U_2\setminus\{u\}} \subset \twoheadrightarrow_{\partial''_2}\twoheadleftarrow_{\partial''_1}$ avec $Roots(\partial''_1, \partial''_2) = Roots((U_1 \setminus \{u\}) \cup (U_2 \setminus \{u\})) = Roots((U_1 \cup U_2) \setminus \{u\})$.

   Or ∂'_1 et ∂'_2 sont dominées par u et $u \parallel Roots(\partial''_1, \partial''_2)$. On peut donc former les dérivations parallèles $\partial'_1 \cup \partial''_1$ et $\partial'_2 \cup \partial''_2$. On a bien $\twoheadleftarrow_{\partial_1}\twoheadrightarrow_{\partial_2} \subset \twoheadrightarrow_{\partial'_2\cup\partial''_2}\twoheadleftarrow_{\partial'_1\cup\partial''_1}$ et $Roots(U_1 \cup U_2) = Roots(\partial'_1, \partial'_2) \cup Roots(\partial''_1, \partial''_2)$.

2. Sinon, considérons le cas où $V_2 = \{u_2 \in U_2 \mid u_2 > u_1\} \neq \varnothing$. Puisque U_2 est constitué d'occurrences disjointes, on peut placer les réductions aux occurrences V_2 en tête de ∂_2 : $\twoheadrightarrow_{\partial_2} = \twoheadrightarrow_{\partial_2|V_2}\twoheadrightarrow_{\partial_2|U_2\setminus V_2}$.

Par application du lemme 5.1, il existe une dérivation ∂'_2 dominée par u_1 telle que $\leftarrow_{\partial_1|u_1} \twoheadrightarrow_{\partial_2|V_2} \subset \twoheadrightarrow_{\partial'_2} \leftarrow_{\partial_1|u_1}$. Donc $\twoheadleftarrow_{\partial_1} \twoheadrightarrow_{\partial_2} \subset \twoheadleftarrow_{\partial_1|U_1\setminus\{u_1\}} \twoheadrightarrow_{\partial'_2} \leftarrow_{\partial_1|u_1} \twoheadrightarrow_{\partial_2|U_2\setminus V_2}$.

D'une part U_1 est constitué d'occurrences disjointes et ∂'_2 est dominée par u_1, donc $U_1 \setminus \{u_1\} \parallel \partial'_2$ et $\twoheadleftarrow_{\partial_1|U_1\setminus\{u_1\}} \twoheadrightarrow_{\partial'_2} = \twoheadrightarrow_{\partial'_2} \twoheadleftarrow_{\partial_1|U_1\setminus\{u_1\}}$.

D'autre part, $u_1 \parallel (U_2 \setminus V_2)$. En effet, si $u_2 \in U_2 \setminus V_2$ vérifie $u_2 \leqslant u_1$ alors $u_2 \leqslant u_1 \leqslant v_2$ pour $v_2 \in V_2 \neq \varnothing$ contredit le fait que U_2 est constitué d'occurrences disjointes. De même, on ne peut pas avoir $u_2 \geqslant u_1$ car le cas $u_2 = u_1$ a déjà été traité et éliminé, et le cas $u_2 > u_1$ contredit $u_2 \notin V_2$. Par conséquent, $\leftarrow_{\partial_1|u_1} \twoheadrightarrow_{\partial_2|U_2\setminus V_2} = \twoheadrightarrow_{\partial_2|U_2\setminus V_2} \leftarrow_{\partial_1|u_1}$.

Par conséquent, $\twoheadleftarrow_{\partial_1} \twoheadrightarrow_{\partial_2} \subset \twoheadrightarrow_{\partial'_2} \twoheadleftarrow_{\partial_1|U_1\setminus\{u_1\}} \twoheadrightarrow_{\partial_2|U_2\setminus V_2} \leftarrow_{\partial_1|u_1}$. On peut appliquer l'hypothèse de récurrence à $\partial_{1|U_1\setminus\{u_1\}}$ et $\partial_{2|U_2\setminus V_2}$. Il existe donc des dérivations $\partial''_1 \in Deriv(R_1)$ et $\partial''_2 \in Deriv(R_2)$ telles que $\twoheadleftarrow_{\partial_1|U_1\setminus\{u_1\}} \twoheadrightarrow_{\partial_2|U_2\setminus V_2} \subset \twoheadrightarrow_{\partial''_2} \twoheadleftarrow_{\partial''_1}$, et dont les racines vérifient $Roots(\partial''_1, \partial''_2) = Roots((U_1 \setminus \{u_1\}) \cup (U_2 \setminus V_2))$.

Or $\partial'_2 \parallel U_1 \setminus \{u_1\}$ et $\partial'_2 \parallel U_2 \setminus V_2$ et $u_1 \parallel U_1 \setminus \{u_1\}$ et $u_1 \parallel U_2 \setminus V_2$. On peut donc former les dérivations parallèles $\partial_{1|u_1} \cup \partial''_1$ et $\partial''_2 \cup \partial'_2$. On a bien $\twoheadleftarrow_{\partial_1} \twoheadrightarrow_{\partial_2} \subset \twoheadrightarrow_{\partial''_2\cup\partial'_2} \twoheadleftarrow_{\partial'_1\cup\partial''_1}$ et $Roots(U_1 \cup U_2) = Roots(u_1, \partial''_1, \partial'_2, \partial''_2)$.

3. Sinon, si $\exists u_2 \in U_2 \;\; u_2 < u_1$, on permute le rôle de ∂_1 et ∂_2 et l'on procède comme au point 2 ci-dessus.

4. Il ne reste que le cas où $u_1 \parallel U_2$. On peut alors permuter $\twoheadleftarrow_{\partial_1} \twoheadrightarrow_{\partial_2} = \twoheadleftarrow_{\partial_1|U_1\setminus\{u_1\}} \rightarrow_{\partial_1|u_1} \twoheadrightarrow_{\partial_2} = \twoheadleftarrow_{\partial_1|U_1\setminus\{u_1\}} \twoheadrightarrow_{\partial_2} \rightarrow_{\partial_1|u_1}$ et appliquer l'hypothèse de récurrence à $\partial_{1|U_1\setminus\{u_1\}}$ et ∂_2. La conclusion de ce cas de figure est identique ensuite à celle du point 2. $\square$

Démonstration (lemme 5.6 page 162). Soit une dérivation $M \leftarrow_{u_0,r_0,\sigma_0} N \twoheadrightarrow_{(u,r_u,\sigma_u)_{u\in U}} P$ avec $r_0 = \langle \alpha_0 \rightarrow \beta_0 \rangle$ et $r_u = \langle \alpha_u \rightarrow \beta_u \rangle$ pour $u \in U$. Par définition, cela signifie $N/u_0 = \sigma_0\alpha_0$, $M = N[u_0 \leftarrow \sigma_0\beta_0]$, et pour tout $u \in U$, $N/u = \sigma_u\alpha_u$, et $P = N[u \leftarrow \sigma_u\beta_u]$.

Comme on l'a vu à propos de la démonstration du lemme 5.1, pour tout $u \in U$ l'occurrence u/u_0 pointe sous une variable x_u de $Var(\alpha_0)$ dans le terme $N/u_0 = \sigma_0\alpha_0$. Autrement dit, il existe une occurrence $v_u \in Occ_{x_u}(\alpha_0)$ telle que $v_u \leqslant u/u_0$. Soit $w_u = (u/u_0)/v_u = u/(u_0 \cdot v_u)$; c'est l'occurrence de la dérivation par r_u dans $\sigma_0 x_u$. Nous avons $\sigma_0 x_u/w_u = N/u = \sigma_u\alpha_u$. Notons $X_U = \{x_u \mid u \in U\}$ l'ensemble des variables de α_0 traversées par U et, pour tout $x \in X$, $U_x = \{u \in U \mid x_u = x\}$, l'ensemble des occurrences de U qui traversent dans N une variable x donnée.

1. Pour tout $x \in X_U$ et tout $u \in U_x$, on note $t_u = \sigma_0(x)[w_u \leftarrow \sigma_u\beta_u]$. On a alors la famille de dérivations $(\sigma_0(x) \rightarrow_{w_u,r_u} t_u)_{u\in U_x}$. D'après le lemme 5.5, il existe un terme t'_x et des dérivations concrètes $(t_u \rightarrow^*_{\nabla_u} t'_x)_{u\in U_x}$. Pour tout $u \in U_x$, on a $\sigma_0(x) \rightarrow^*_{(w_u,r_u,\sigma_u);\nabla_u} t'_x$. On choisit n'importe quel élément $u_x \in U_x \neq \varnothing$ pour définir $\nabla' = \bigcup_{x\in X_U} u_0 \cdot Occ_x(\beta_0) \cdot ((w_{u_x}, r_{u_x}, \sigma_{u_x}); \nabla_{u_x})$. On a alors $M \rightarrow^*_{\nabla'} M' = M[u_0 \cdot Occ_x(\beta_0) \leftarrow t'_x \mid x \in X_U]$.

2. Nous avons fait en sorte que, pour $x \in X_U$ fixé, chacun des termes $\sigma_0 x$ sous la variable x de β_0 dans M subisse la même dérivation. Nous pouvons ainsi former une substitution θ sur les variables de α_0 et β_0 par $\theta(x) = t'_x$ si $x \in X_U$ et $\theta(x) = \sigma_0(x)$ si $x \notin X_U$. On a alors $\theta\beta_0 = \theta\beta_0[Occ_x(\beta_0) \leftarrow \theta x \mid x \in Var(\beta_0)] = \sigma_0\beta_0[Occ_x(\beta_0) \leftarrow t'_x \mid x \in X_U] = M/u_0[Occ_x(\beta_0) \leftarrow t'_x \mid x \in X_U] = M'/u_0$. Par conséquent, $M' \leftarrow_{u_0,r_0,\theta} N'$ avec $N' = M'[u_0 \leftarrow \theta\alpha_0]$.

3. On cherche à atteindre N' en partant de $P = N[u_0 \cdot v_u \cdot w_u \leftarrow \sigma_u\beta_u \mid u \in U]$. Remarquons que $N' = M'[u_0 \leftarrow \theta\alpha_0] = M[u_0 \leftarrow \theta\alpha_0] = N[u_0 \leftarrow \theta\alpha_0] = N[u_0 \cdot Occ_x(\alpha_0) \leftarrow t'_x \mid x \in X_U]$. Or, pour

tout $x \in X_U$, on a $N/u_0 \cdot Occ_x(\alpha_0) = \sigma_0(x)$, et $\sigma_0(x) \to^*_{(w_u,r_u,\sigma_u);\nabla_u} t'_x$ pour tout $u \in U_x$. On définit $\nabla' = \prod_{x \in X_U} u_0 \cdot (\prod_{u \in U_x} v_u \cdot \nabla_u)$. On a alors $N \twoheadrightarrow_\partial P \to^*_{\nabla'} N'$.

En conclusion, il existe bien $M \to^*_\nabla M' \leftarrow_{\partial_0} N' \leftarrow^*_{\nabla'} P$. $\square$

D.6 Algorithme et terminaison.

Démonstration (lemme 6.2 page 184). On démontre cette propriété par récurrence structurelle sur s et t, suivant les deux cas possibles pour la condition $s \trianglelefteq t$.

- Si $s \trianglelefteq t_i$ alors $|s| \leqslant |t_i|$ par hypothèse de récurrence, or $|t_i| \leqslant \sum_{k=1}^n |t_k| = |t| - 1$, donc $|s| \leqslant |t|$.
- Si $\forall j \in [n]\ \ s_j \trianglelefteq t_{i_j}$ alors $|s_j| \leqslant |t_{i_j}|$ par hypothèse de récurrence, or $|t_{i_j}| \leqslant \sum_{k=1}^n |t_k| = |t| - 1$, donc $\forall j \in [n]\ \ |s_j| \leqslant |t| - 1$, donc $|s| - 1 \leqslant \sum_{j=1}^n |s_j| \leqslant |t| - 1$, donc $|s| \leqslant |t|$.

Si t est un sous-terme propre de s, alors $|s| \not\leqslant |t|$. On a donc $s \ntrianglelefteq t$ par contraposition de la proposition précédente. Cette propriété est vrai également pour la hauteur des termes, pour les même raisons.

Annexe N

Notations, Définitions et Propositions Usuelles

Nous donnons ici les notations, définitions et propositions usuelles que nous ne rappelons pas explicitement dans le texte. Pour plus de détails, on peut consulter par exemple [HO80, Wir90] pour les algèbres, [Hue80] pour les occurrences, [Cou83] pour les arbres, [GTWW77] pour les algèbres initiales et continues, [Der87, Rao89] pour les ensembles (pré)ordonnés, [GS90] pour les ensembles ordonnés complets, [Hue76, Ede85] pour les substitutions, [Hue80] pour les notions de réduction et de confluence, et [HO80, DJ90] pour les systèmes de réécriture.

N.1 Ensembles.

Soit E un ensemble.

- $\mathfrak{P}(E)$ est l'ensemble des parties de E
- $E^* = \bigcup_{n\in \mathbb{N}} E^n$
- $E^+ = \bigcup_{n\in \mathbb{N}\setminus\{0\}} E^n$
- $\complement_E A = E \setminus A$ est le complémentaire de A dans E (on omet souvent l'indice E)
- $|E|$ est le cardinal de E
- La notation informelle $(x_1, \ldots, x_n)$ désigne une suite de n éléments, vide dans le cas $n = 0$
- $\forall a, b \in \mathbb{N} \quad [a..b] = \{x \in \mathbb{N} \mid a \leqslant x \leqslant b\},\ [a..b[= \{x \in \mathbb{N} \mid a \leqslant x < b\},\ [b] = [1..b],\ [a.. + \infty[= \{x \in \mathbb{N} \mid a \leqslant x\}$
- $\mathbb{R}_+ = [0, +\infty[$
- $\forall x \in \mathbb{R} \quad \lfloor x \rfloor$ est la partie entière de x.

N.2 Relations.

Soit $\lhd$ une relation sur E.

- Pour tout $A, B \subset E$ on note $A \lhd B$ ssi $\forall a \in A \quad \forall b \in B \quad a \lhd b$
- $\lhd_1 \lhd_2$ est la relation définie par : $\forall x, z \in E \quad x\ (\lhd_1 \lhd_2)\ z$ ssi $\exists y \in E \quad x \lhd_1 y \lhd_2 z$

N.3 Fonctions.

Soient E et F des ensembles.

- $E \to F$ (resp. F^E) est l'ensemble des fonctions (resp. applications) de E dans F
- On note $Dom(f)$ est le domaine d'une fonction f et $Im(f)$ son image.
- $f_{|A}$ est la restriction d'une fonction f à un ensemble A : $Dom(f_{|A}) = Dom(f) \cap A$ et $\forall x \in Dom(f_{|A})$ $f_{|A}(x) = f(x)$
- $E \overset{\text{fin}}{\to} F$ est l'ensemble des fonctions de E dans F, de domaine fini
- $Id_E : E \to E$ est la fonction identité (l'indice est omis en pratique)

Si E a un élément particulier, noté 0 (l'élément nul),

- Une famille $(x_i)_{i\in I}$ de E^I est *presque nulle* ssi $\{i \in I \mid x_i \neq 0\}$ est fini.
- $E^{(I)}$ est l'ensemble des familles finies

N.4 Algèbre et magma.

Le terme de F-*magma*, ainsi que la notation $M(F)$, est fréquemment employé dans le cadre des schémas de programme. Les termes plus courant de Σ-*algèbre* ou de F-*algèbre* en sont d'exacts synonymes.

- Un ensemble de *sortes* est un ensemble fini $\mathcal{S}$.
- L'ensemble des *types du premier ordre* sur $\mathcal{S}$ est $\mathcal{S}^+$.
  - Un type $s \in \mathcal{S}^1$ est un *type de constante* d'*arité* nulle.
  - Un type $(s_1, \ldots, s_n, s) \in \mathcal{S}^{n+1}$ est un *type d'opérateur* d'*arité* n, noté $s_1 \times \cdots \times s_n \to s$.
- Une $\mathcal{S}$-*signature* est un couple (F, τ).
  - F est un ensemble de *noms d'opérateur*.
  - $\tau : F \to \mathcal{S}^+$ est une *fonction de typage*.

  La notation (F, τ) est abrégée en F, et expansée en $\{f : \tau(f) \mid f \in F\}$. L'ensemble F_n désigne le sous-ensemble des opérateurs de F d'arité n. Pour tout $f \in F$, on définit $\sigma(f) = s$ pour $\tau(f) = s_1 \times \cdots \times s_n \to s$.
- La notation abrégée $f : s^* \to s'$ définit une famille $(f_n)_{n\in\mathbb{N}}$ d'opérateurs typés $\tau(f_n) = s^n \to s$.
- Une F-*algèbre* (ou F-*magma*) est une structure $\boldsymbol{M} = \langle (M_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{M}})_{f\in F}\rangle$.
  - $(M_s)_{s\in\mathcal{S}}$ est une famille d'ensembles indexés par $\mathcal{S}$.
  - Pour tout $f \in F$ de type $s_1 \times \cdots \times s_n \to s$, $f_{\boldsymbol{M}}$ est une application de $M_{s_1} \times \cdots \times M_{s_n} \to M_s$.
- Une F'-algèbre $\boldsymbol{M}'$ est une *extension* d'une F-algèbre $\boldsymbol{M}$ ssi :
  - $\mathcal{S} \subset \mathcal{S}'$ et $\forall s \in \mathcal{S}$ $M'_s = M_s$
  - $F \subset F'$ et $\forall f \in F$ $f_{\boldsymbol{M}'} = f_{\boldsymbol{M}}$

Nous omettons souvent l'indication des sortes afin d'alléger les notations.

N.5 Occurrence.

- L'ensemble des occurrences u est $Occ = (\mathbb{N} \setminus \{0\})^*$; c'est une séquence finie d'entiers strictement positifs.

- L'*occurrence vide* est notée ϵ.
- La *concaténation* des occurrences u et v est notée $u{\cdot}v$ (ou uv lorsque le contexte l'autorise).
- Si U et V sont des ensembles d'occurrences, $U{\cdot}V = \{u{\cdot}v \mid u \in U, v \in V\}$
- $(Occ, \cdot, \epsilon)$ forme un monoïde.
- Occ est partiellement ordonné par la relation de *domination* $\leqslant$ suivante.
  - $u \leqslant v$ ssi $\exists w \quad v = u{\cdot}w$. On note alors $w = v/u$ et l'on dit que u *est au dessus de* v.
  - $u < v$ ssi $u \leqslant v$ et $u \neq v$
  - $u \leqslant V$ ssi $\forall v \in V \quad u \leqslant v$ et $U \leqslant v$ ssi $\forall u \in U \quad u \leqslant v$ et $U \leqslant V$ ssi $\forall u \in U \quad \forall v \in V \quad u \leqslant v$.
- Deux occurrences u et v sont *disjointes* (ou *parallèles*), noté $u \parallel v$, ssi $u \nleqslant v$ et $v \nleqslant u$.
- Une occurrence u est disjointe d'un ensemble d'occurrences V, noté $u \parallel V$, ssi $\forall v \in V \quad u \parallel V$.
- Deux ensembles d'occurrences U et V sont *parallèles*, noté $U \parallel V$, ssi $\forall u \in U \quad \forall v \in V \quad u \parallel v$.
- L'ensemble U est un ensemble d'occurrences disjointes ssi $\forall u \in U \quad \forall u' \in U \quad u \neq u' \Rightarrow u \parallel u'$.

N.6 Arbre.

- Un *arbre* t sur la signature F est une fonction de $Occ \to F$ dont le domaine est un *domaine d'arbre* (voir [Cou83]). On note $f(t_1, \ldots, t_n)$ l'arbre t de nœud f qui a pour fils les *sous-arbres* $t_1, \ldots, t_n$. Il est *de sorte* $\tau(t) = s$ ssi $\tau(f) = s_1 \times \cdots \times s_n \to s$. Il est *bien formé* ssi $t_1, \ldots, t_n$ sont bien formés et de sorte respective $s_1, \ldots, s_n$.
- $Dom(t)$ est le *domaine* de l'arbre t ; c'est un ensemble d'occurrences. L'arbre t est *fini* ssi $Dom(t)$ est fini. Il est *infini* dans le cas contraire.
- $t(u)$ est le symbole de F à l'occurrence u de l'arbre t.
- t/u est le sous-arbre de t à l'occurrence u.
- $t[u \leftarrow t']$ est l'arbre t où l'on a remplacé à l'occurrence u le sous-arbre t/u par l'arbre t'.
- $t[u \leftarrow \Box]$ est un *contexte*, fonction qui à t' associe $t[u \leftarrow t']$.
- $t[x_1/t_1, \ldots, x_n/t_n]$ est l'arbre t où l'on a remplacé le sous-arbre x_i par l'arbre t_i à chaque occurrence du symbole x_i, pour tout $i \in [n]$.
- La *taille* d'un arbre t est le cardinal de son domaine : $|t| = |Dom(t)|$. C'est aussi son nombre de nœuds. On a également $|f(t_1, \ldots, t_n)| = 1 + |t_1| + \cdots + |t_n|$.
- $Occ_g(t)$ est l'ensemble des occurrences du symbole g dans l'arbre t et $Occ_G(t) = \bigcup_{g \in G} Occ_g(t)$
- $Roots(U) = \mathrm{Emin}_{\leqslant}(U) = \{u \in U \mid \forall v \in U \quad v \leqslant u \Rightarrow u \leqslant v\}$ est l'ensemble des racines d'un ensemble d'occurrences U.
- $M_s(F)$ est l'ensemble des arbres finis de sorte s sur F.
- $M(F) = \bigcup_{s \in \mathcal{S}} M_s(F)$ est l'ensemble des arbres finis sur F. On note aussi $M(F) = (M_s(F))_{s \in \mathcal{S}}$.
- $M^\infty(F)$ est l'ensemble de tous les arbres (finis ou infinis) sur F.
- $M_\Omega(F) = M(F \cup \Omega)$ où $\Omega = \{\Omega_s : s\}_{s \in \mathcal{S}}$ est une signature disjointe de F.
- $M_\Omega^\infty(F) = M^\infty(F \cup \Omega)$.

Lorsque nous omettons l'indication des sortes, Ω désigne aussi l'un des symboles Ω_s.

- Lemme de König : tout arbre infini tel que chaque nœud n'a qu'un nombre fini de fils contient un chemin (une branche) infini.

N.7 Algèbre des termes.

Soit F une signature.

- $\boldsymbol{M}(F)$ est la F-*algèbre des termes du premier ordre*, constituée des domaines $(M_s(F))_{s\in\mathcal{S}}$ et des fonctions de construction d'arbres associées.
- $X = (X_s)_{s\in\mathcal{S}}$ est une famille d'ensembles de *variables*.
- $\boldsymbol{M}(F,X) = \boldsymbol{M}(F \cup X)$ est la F-*algèbre libre* engendrée par X.
- $Var(t)$ est l'ensemble des variables de X qui apparaissent dans le terme t.
- Un terme t est *linéaire* ssi chaque variable x de $Var(t)$ n'a qu'une occurrence dans t. Il est *d'ordre zéro* s'il n'a aucune occurrence de variable.

N.8 Algèbre initiale.

- Une F-algèbre $\boldsymbol{M}$ est *initiale* dans une classe d'algèbre ssi pour toute F-algèbre $\boldsymbol{M}'$ dans cette classe il existe un unique homomorphisme $\boldsymbol{eval}_{\boldsymbol{M}'} : \boldsymbol{M} \to \boldsymbol{M}'$.
- L'algèbre $\boldsymbol{M}(F)$ des termes du premier ordre est initiale. Pour toute F-algèbre $\boldsymbol{M}$, on note $t_{\boldsymbol{M}} = \boldsymbol{eval}_{\boldsymbol{M}}(t)$ l'élément de $\boldsymbol{M}$ dénoté par le terme t.
- $\forall t \in M(F,X)$ tq $Var(t) \subset X_n = (x_1,\ldots,x_n)$, on note $t_{X_n,\boldsymbol{M}}$ la fonction de $M^n \to M$ tq $t_{X_n,\boldsymbol{M}}(m_1,\ldots,m_n) = t[x_1/m_1,\ldots,x_n/m_n]_{\boldsymbol{M}}$.

N.9 Ensemble muni d'une relation.

Soit E un ensemble muni d'une relation $\preccurlyeq$ (dont on note la réciproque $\succcurlyeq$).

- La *relation stricte* $\prec$ associée à $\preccurlyeq$ est définie par $\forall x,y \in E \quad x \prec y$ ssi $x \preccurlyeq y$ et $x \not\succcurlyeq y$
- La relation $\preccurlyeq$ est *bien fondée* ssi il n'existe pas dans E de suite infinie strictement décroissante (pour $\prec$)

Soit A une partie de E.

- L'ensemble des *éléments minimaux de* A est $\mathrm{Emin}_{\preccurlyeq}(A) = \{x \in A \mid \forall y \in A \quad y \preccurlyeq x \;\Rightarrow\; x \preccurlyeq y\}$
- L'ensemble des *plus petits éléments de* A est $\mathrm{Min}_{\preccurlyeq}(A) = \{x \in A \mid \forall y \in A \quad x \preccurlyeq y\}$
- L'ensemble des *minorants de* A *dans* E est $\mathrm{Minor}_{E,\preccurlyeq}(A) = \{x \in E \mid \forall y \in A \quad x \preccurlyeq y\}$
- L'ensemble des *bornes inférieures de* A *dans* E est $\mathrm{Inf}_{E,\preccurlyeq}(A) = \mathrm{Max}_{\preccurlyeq}(\mathrm{Minor}_{E,\preccurlyeq}(A)) =$
  $\{x \in E \mid \forall y \in A \quad x \preccurlyeq y$ et $\forall z \in E \quad (\forall y \in A \quad z \preccurlyeq y) \;\Rightarrow\; z \preccurlyeq x\}$ avec $\mathrm{Max}_{\preccurlyeq}(A) = \mathrm{Min}_{\succcurlyeq}(A)$

On définit de même (en considérant la relation réciproque) les *éléments maximaux*, les *plus grands éléments*, les *majorants*, et les *bornes supérieures*. Ces définitions s'emploient en fait le plus souvent pour des préordres ($\preccurlyeq$ est transitive et réflexive) ou des ordres ($\preccurlyeq$ est de plus antisymétrique).

N.10 Ensemble préordonné.

Soit $(E,\preccurlyeq)$ un ensemble préordonné.

- La relation $\preccurlyeq$ est un *beau préordre* ssi
  - $\forall (x_n)_{n\in N} \in E^{\mathbb{N}} \quad \exists\, i < j \in \mathbb{N} \quad x_i \preccurlyeq x_j$
- La relation $\preccurlyeq$ est un *bon préordre* (well-quasi-order) ssi c'est un beau préordre total.

- Le *plongement* $\trianglelefteq_{\preccurlyeq}$ de $\preccurlyeq$ dans les séquences (embedding) est défini par :
  - $\forall n \leqslant m \in \mathbb{N} \quad \forall x_1, \ldots, x_n, y_1, \ldots, y_m \in E \quad (x_1, \ldots, x_n) \trianglelefteq_{\preccurlyeq} (y_1, \ldots, y_m)$ ssi
    $\exists\, 1 \leqslant i_1 < \cdots < i_n \leqslant m \quad \forall j \in [m] \quad x_{i_j} \preccurlyeq y_j$
- Lemme de Higman [Hig52, NW63] : le plongement $\trianglelefteq_{\preccurlyeq}$ est un beau préordre ssi $\preccurlyeq$ est un beau préordre.
- Le *plongement homéomorphe* $\trianglelefteq_{\preccurlyeq}$ (homeomorphic embedding) d'une signature F dans les arbres finis $M(F)$ est défini par :
  - $\forall s, t \in M(F) \quad s = f(s_1, \ldots, s_m) \trianglerighteq_{\preccurlyeq} g(t_1, \ldots, t_n) = t$ ssi
    $(\exists i \in [m] \quad s_i \trianglerighteq_{\preccurlyeq} t)$ ou $(f \preccurlyeq g$ et $\exists i_1, \ldots, i_n \in \mathbb{N} \quad 1 \leqslant i_1 < \cdots < i_n \leqslant m$ tq $\forall j \in [n]$ $s_{i_j} \trianglerighteq_{\preccurlyeq} t_j)$

  Lorsque la relation $\preccurlyeq$ est l'égalité, on parle de plongement homéomorphe pur (pure homeomorphic embedding).
- Théorème de Kruskal [Kru60, NW63] : le plongement homéomorphe $\trianglelefteq_{\preccurlyeq}$ est un beau préordre sur $M(f)$ ssi $\preccurlyeq$ est un beau préordre sur F.

Par exemple, la relation d'ordre usuelle sur les entiers naturels est un bon ordre.

N.11 Ensemble ordonné.

Soient $(E, \leqslant)$ un ensemble ordonné et $A \subset E$

- S'il existe une *borne inférieure* (resp. *supérieure*), elle est unique et notée $\inf(A)$ (resp. $\sup(A)$)
- On note $\bot_E = \inf(E)$ (resp. $\top_E = \sup(E)$ le plus petit (resp. plus grand) élément de E, lorsqu'il existe.

À la différence de la terminologie anglaise, la relation $\leqslant$ n'est pas nécessairement totale ; pour insister sur ce point, on parle d'*ensemble partiellement ordonné*.

N.12 Ensemble ordonné complet.

- Un *ensemble partiellement ordonné* (poset) est un ensemble D muni d'une relation d'ordre partielle $\leqslant$.
- L'ordre $\leqslant$ est *plat* ssi $\forall d, d' \in D \quad d \leqslant d' \;\Leftrightarrow\; d = d' \;\vee\; d = \bot_D$. On construit un *domaine primitif* $(D, \leqslant)$ sur un ensemble E quelconque en lui ajoutant un nouvel élément $\bot$ et en définissant $\leqslant$ comme un ordre plat. On note $D = E_\bot = E \cup \{\bot\}$.
- Un sous-ensemble Δ de D est *dirigé* (directed) ssi chaque sous-ensemble fini propre de Δ admet une borne supérieure.
- Un ensemble partiellement ordonné $(D, \leqslant)$ est *complet* (cpo) ssi tout sous-ensemble dirigé Δ de D admet une borne supérieure. Nous supposons également que $(D, \leqslant)$ admet un plus petit élément $\bot$ (voir [GTWW77] pour différentes notions de complétude). On dit aussi que $(D, \leqslant)$ est un *domaine* (voir néanmoins [GS90]).
- Soient $(D, \leqslant)$ et $(D', \leqslant')$ deux cpos et une f application de $D \to D'$.
  - f est *monotone* ssi $\forall x, y \in D \quad x \leqslant y \;\Rightarrow\; f(x) \leqslant' f(y)$
  - f est *continue* ssi pour tout sous-ensemble dirigé Δ de D, $f(\sup(\Delta)) = \sup(f(\Delta))$.
- Si $(D_i, \leqslant_i)_{i \in [n]}$ est une famille d'ensembles partiellement ordonnés, l'ensemble produit $D = D_1 \times \cdots \times D_n$ est partiellement ordonné par l'*ordre partiel produit* $\leqslant$ défini ainsi :
  - $\forall (x_i)_{i \in [n]}, (y_i)_{i \in [n]} \in D \quad (x_1, \ldots, x_n) \leqslant (y_1, \ldots, y_n)$ ssi $\forall i \in [n] \quad x_i \leqslant_i y_i$.

  Si les $(D_i, \leqslant_i)$ sont complets, alors $(D, \leqslant)$ l'est aussi.

- Proposition : Si E est un ensemble quelconque, alors $(\mathfrak{P}(E), \subset)$ est un ensemble partiellement ordonné complet : pour tout $A \in \mathfrak{P}(E)$, $\inf(A) = \bigcap_{B \in A} B$, $\sup(A) = \bigcup_{B \in A} B$, $\perp = \varnothing$ et $\top = \mathfrak{P}(E)$. De plus, la réunion et l'intersection sont continues de $(\mathfrak{P}(E), \subset)^2$ dans $(\mathfrak{P}(E), \subset)$.
- Théorème du point fixe : Si f est une fonction continue sur un ensemble partiellement ordonné complet D, alors f admet un *plus petit point fixe* $\mathrm{lfp}(f) = \sup_{n \in \mathbb{N}}(f^n(\perp))$, que l'on trouve parfois aussi noté $f{\uparrow}\omega$.

N.13 Algèbre continue.

- Une *F-algèbre ordonnée* $\boldsymbol{D} = \langle (D_s)_{s\in\mathcal{S}}, (\leqslant_s)_{s\in\mathcal{S}}, (\perp_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{D}})_{f\in F}\rangle$ est telle que :
  - $\langle (D_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{D}})_{f\in F}\rangle$ est une F-algèbre.
  - Pour tout $s \in \mathcal{S}$, $(D_s, \leqslant_s)$ est un ensemble partiellement ordonné de plus petit élément $\perp_s$.
  - Les fonctions $(f_{\boldsymbol{D}})_{f\in F}$ sont monotones.
- Une F-algèbre ordonnée est *continue* ssi
  - Pour tout $s \in \mathcal{S}$, l'ensemble partiellement ordonné $(D_s, \leqslant_s)$ est complet.
  - Les fonctions $(f_{\boldsymbol{D}})_{f\in F}$ sont continues.

  Une algèbre *discrète* est telle que tous ses ordres sont plats ; c'est un cas particulier d'algèbre continue.
- $\boldsymbol{M}_\Omega(F) = \langle M_\Omega(F), (\leqslant_s)_{s\in\mathcal{S}}, (\Omega_s)_{s\in\mathcal{S}}, (f_{\boldsymbol{M}})_{f\in F\cup\Omega}\rangle$ est la *F-algèbre ordonnée initiale* ; les $(\leqslant_s)_{s\in\mathcal{S}}$ sont définis précisément comme les plus petits ordres tels que $\boldsymbol{M}_\Omega(F)$ forme une algèbre ordonnée (voir [Cou86, Cou90a]).
- $\boldsymbol{M}_\Omega(F, X) = \boldsymbol{M}_\Omega(F \cup X)$ est la *F-algèbre ordonnée libre* engendrée par X.
- $\boldsymbol{M}_\Omega^\infty(F, X)$ est la *F-algèbre continue libre* engendrée par X.
- Une F'-algèbre continue $\boldsymbol{M}'$ est une *extension* d'une F-algèbre continue $\boldsymbol{M}$ ssi :
  - $\mathcal{S} \subset \mathcal{S}'$ et $\forall s \in \mathcal{S}\ \ (M'_s, \leqslant'_s, \perp'_s) = (M_s, \leqslant_s, \perp_s)$
  - $F \subset F'$ et $\forall f \in F\ \ f_{\boldsymbol{M}'} = f_{\boldsymbol{M}}$

N.14 Schémas de programme récursifs.

Les définitions et propositions relatives aux schémas de programme sont données explicitement dans le texte à la section §2.5.2. Elles sont complétées à la section §5.1.2 pour les questions liées aux classes d'interprétation, à la comparaison de schémas, aux classes équationnelles, et à la correction des transformations. Elles sont suivies à la section §5.2.1 de la définition des transformations usuelles basées sur le pliage et le dépliage. La section §5.5.1 donne également les définitions qui concernent les inconnues auxiliaires.

N.15 Réduction et confluence.

Soit $\rightarrow$ une relation sur un ensemble E, dite *relation de réduction*.

- $\rightarrow^{-1} = \{(x, y) \in E \mid y \rightarrow x\}$ est la relation réciproque
- La composition des réductions $\rightarrow_1$ et $\rightarrow_2$ est $\rightarrow_1 \rightarrow_2 = \{(x, z) \in E \mid \exists y \in E\ \ x \rightarrow_1 y \rightarrow_2 z\}$
- $\rightarrow^0 = \{(x, x) \in E \mid x \in E\}$ est l'égalité
- $\rightarrow^{n+1} = \rightarrow^n \rightarrow$ pour $n \in \mathbb{N}$

- $\to^* = \bigcup_{n\in\mathrm{IN}} \to^n$ est la fermeture réflexive-transitive de $\to$
- $\to^+ = \bigcup_{n\in\mathrm{IN}\setminus\{0\}} \to^n$ est la fermeture transitive de $\to$
- $\leftrightarrow\ =\ \to \cup \leftarrow$ est la fermeture symétrique de $\to$
- Une *dérivation* finie est une suite finie de réductions $x_0 \to \ldots \to x_n$. Une dérivation infinie est une suite infinie de réductions.
- La relation $\to$ est *confluente* ssi $\forall x, y, z \in E \quad x \leftarrow^* z \to^* y \;\Rightarrow\; \exists t \in E \quad x \to^* t \leftarrow^* y$

N.16 Substitution.

En matière de substitutions, on rencontre plus souvent la notation V pour l'ensemble des variables, ici noté X.

- Une *substitution* est une application $\sigma : X \to M(F, X)$ telle que l'ensemble $Dom(\sigma) = \{x \in X \mid \sigma(x) \neq x\}$ soit fini ; $Dom(\sigma)$ est le *domaine* de σ. Nous notons $\{x_1/t_1, \ldots, x_n/t_n\}$ la substitution σ ; les x_i sont les éléments de $Dom(\sigma)$ et les termes $t_i = \sigma(x_i)$ forment l'*image* $Im(\sigma)$.
- La *substitution identité* (ou *vide*) est notée ε.
- Les substitutions sont étendues par morphisme à $M(F, X)$: $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. Si T est un ensemble de termes, on note $\sigma(T) = \{\sigma(t) \mid t \in T\}$.
- La *composition* $\theta \circ \sigma$ de deux substitutions θ et σ est notée $\theta\sigma$.
- On note $Subst(F, X)$ l'*ensemble des substitutions*. Muni de la composition et de la substitution identité, il forme un monoïde non-commutatif.
- La *restriction* $\sigma\!\upharpoonright\! V$ de σ à un sous-ensemble de variables $V \subset X$ est définie par $(\sigma\!\upharpoonright\! V)(x) =$ si $x \in V$ alors $\sigma(x)$ sinon x. C'est aussi une substitution.
- Un *renommage* (ou *permutation*) η est une substitution qui envoye X sur X de manière bijective. On note $Ren(X)$ l'ensemble des renommages.
- Soit $\leqslant$ la relation (subsumption quasi-ordering) sur les termes définie par $t \leqslant t'$ ssi il existe une substitution σ telle que $t' = \sigma(t)$. C'est un préordre sur $M(F, X)$. Si une telle σ existe, la substitution $\sigma\!\upharpoonright\! Var(t)$ est unique ; on la note $t' \div t$.
- Soit $\equiv$ la relation sur les termes définie par $t \equiv t'$ ssi $t \leqslant t'$ et $t \geqslant t'$. C'est une relation d'équivalence sur $M(F, X)$. Elle vérifie $t \equiv t'$ ssi il existe une permutation η telle que $t = \eta(t')$.
- Une substitution θ est *plus générale* qu'une substitution σ, noté $\theta \leqslant \sigma$, ssi il existe une substitution γ telle que $\gamma\theta = \sigma$. La relation $\leqslant$ est un préordre sur $Subst(F, X)$.
- Soit S un ensemble de termes. Un *unificateur* de S est une substitution θ telle que $\theta(S)$ est un singleton. On dit alors que S est *unifiable* et que l'élément de $\theta(S)$ est une *unification* de S. Un unificateur θ est un *unificateur principal* de S ssi pour tout unificateur σ de S, $\theta \leqslant \sigma$. On note $Mgu(S)$ l'ensemble des unificateurs principaux de S. Nous avons les propositions suivantes.
  - Si S admet un unificateur, alors il admet un unificateur principal.
  - Si $\theta \in Mgu(S)$, alors $Mgu(S) = Ren(X)\theta$.

N.17 Système de réécriture.

Dans le cadre des systèmes de réécriture, les termes sont souvent notés M, N, P, Q plutôt que t.

- Un *système de réécriture* R est un sous-ensemble de paires (α, β) de $M(F, X) \times M(F, X)$, appelées *règles* et notées $r = \langle \alpha \to \beta \rangle$. On note $r^{-1} = \langle \beta \to \alpha \rangle$ la *règle réciproque* et R^{-1} le système de réécriture $\{\langle \beta \to \alpha \rangle \mid \langle \alpha \to \beta \rangle \in R\}$.

- Une *réduction de M en N à l'occurrence u par la règle r au moyen de la substitution σ* est un quintuplet (M, u, r, σ, N), noté $M \rightarrow_{u,r,\sigma} N$ tel que $M/u = \sigma\alpha$ et $N = [u \leftarrow \sigma\beta]$. La *relation de réduction $\rightarrow_R$* sur les termes est définie par $M \rightarrow_R N$ ssi il existe une réduction $M \rightarrow_{u,r,\sigma} N$.
- u est l'occurrence d'une *expression réductible* (redex) de R dans le terme M ssi $u \in Occ(M)$ et il existe une règle $r = \langle \alpha \rightarrow \beta \rangle$ de R telle que $\alpha \leqslant M/u$. Elle détermine une unique réduction par $\sigma = (M/u) \div \alpha$ et $N = M[u \leftarrow \sigma(\beta)]$. Cette définition n'a vraiment de sens que pour $Var(\alpha) \supset Var(\beta)$.
- Un système de réécriture est *linéaire* (resp. d'ordre zéro) *à gauche* (resp. *à droite*) ssi pour tout $\langle \alpha \rightarrow \beta \rangle$ de R, le terme α (resp. β) est linéaire (resp. d'ordre zéro) *à gauche* (resp. *à droite*). Il est *linéaire* ssi il est linéaire à gauche et à droite.

À l'instar de [Cou86, DJ90, Cou90a], et au contraire de [Hue80, HO80], nous n'imposons pas dans la définition des *systèmes de réécriture* que les règles $\langle \alpha \rightarrow \beta \rangle$ vérifient $Var(\alpha) \supset Var(\beta)$. Cette contrainte n'intervient que pour assurer le déterminisme d'une réduction étant données une règle et une occurrence ; elle est indispensable lorsque le système de réécriture sert précisément à modéliser un calcul déterministe, comme celui de la sémantique des schémas de programme, des définitions de fonctions primitives récursives, etc. La section §5.3 nécessite quelques définitions supplémentaires (dérivations parallèles, superposition, paire critique) qui ne sont données qu'à ce moment-là.

Références bibliographiques

[AC90] Isabelle Attali and Jacques Chazarain. Functional evaluation of natural semantics specifications. Rapport de Recherche RR-1218, INRIA, France, Mai 1990.

[AG92] L. O. Andersen and C.K. Gomard. Speedup analysis in partial evaluation: preliminary results. In PEPM'92 [PEP92], pages 1–7.

[AK90] Hassan Aït-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Laboratory, Janvier 1990.

[AL93] Andrea Asperti and Cosimo Laneve. Optimal reductions in interaction systems. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Conference on Theory and Practice of Software Development, Orsay, France, April 1993*, LNCS 668, pages 485–500. Springer-Verlag, 1993.

[Amt91] T. Amtoft Hansen. Properties of unfolding-based meta-level systems. In PEPM'91 [PEP91], pages 243–254.

[And92] L. O. Andersen. Self-applicable C program specialization. In PEPM'92 [PEP92], pages 54–61.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison-Wesley, 1986.

[Att89] Isabelle Attali. *Compilation de programmes TYPOL par attributs sémantiques*. Thèse de doctorat, Université de Nice, France, Avril 1989.

[Aug85] L. Augustsson. Compiling pattern matching. In *Proc. Conference on Functional Programming languages and Computer Architecture, Nancy, France, September 1985*, LNCS 201. Springer-Verlag, 1985.

[AvE82] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *J. Association for Computing Machinery*, 29(3):841–862, Juillet 1982.

[BAK91] R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mecanism for logic programs. *J. Theoretical Computer Science*, 86(1):35–79, 1991.

[BC82] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *J. Theoretical Computer Science*, 20:265–321, 1982.

[BCD$^+$88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In 3^{rd} *Symposium on Software Development Environments, Boston, Nov. 1988*, SIGSOFT, Novembre 1988. Aussi publié comme Rapport de Recherche INRIA RR-777, Décembre 1987.

[BD76] R.M. Burstall and J. Darlington. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Association for Computing Machinery*, 24(1):44–67, Janvier 1977.

[BDG88] J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural complexity*, volume I. Springer-Verlag, 1988.

[BEJ88] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial evaluation and mixed computation*. Proc. of the Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark, October 1987. North-Holland, 1988.

[Ber81] Gérard Berry. Programming with concrete data structures and sequential algorithms. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 49–57. ACM, 1981.

[Ber90] A. A. Berlin. Partial evaluation applied to numerical computation. In M. Wand, editor, *Proc. ACM Conference on Lisp and Functional Programming, Nice, France, June 1990*, pages 139–150. ACM Press, Juin 1990.

[BL79] G. Berry and J.-J. Levy. Minimal and optimal computations of recursive programs. *J. Association for Computing Machinery*, 26(1):148–175, 1979.

[Bol91] R. N. Bol. Loop checking in partial deduction. Technical Report CS-R9134, CWI, Amsterdam, Juillet 1991.

[Bon88] A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In Bjørner et al. [BEJ88], pages 27–50.

[Bon91] A. Bondorf. Automatic autoprojection of higher order recursive equations. *J. Science of Computer Programming*, 17, 1991.

[Bon93] A. Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, Denmark, Mai 1993.

[Bul84] M. A. Bulyonkov. Polyvariant mixed computation for analyser program. *Acta Informatica*, 21:473–484, 1984.

[BW90] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. Technical Report CSL-TR-90-422, Stanford University, Ca, Mars 1990.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. DIKU, University of Copenhagen, Denmark, submitted for publication, Octobre 1993.

[Car84] L. Cardelli. Compiling a functional language. In *Proc. Conference on Lisp and Functional Programming, Austin, Texas, August 1984*, 1984.

[CD89] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[CD91a] C. Consel and O. Danvy. For a better support of static data flow. In Hugues [Hug91], pages 496–519.

[CD91b] C. Consel and O. Danvy. Static and dynamic semantics processing. In *18 th Symposium on Principles of Programming Languages, Orlando, Florida, Jan 1991*, pages 14–24. ACM Press, 1991.

[CD93] C. Consel and O. Danvy. Partial evaluation: principles and perspectives. In *Journées francophones des langages applicatifs, Annecy*, Février 1993.

[CK91a] C. Consel and S. C. Khoo. Parametrized partial evaluation. In *Conference on Programming Languages, Design and Implementations*, SIGPLAN Notices, pages 92–106. ACM Press, 1991.

[CK91b] C. Consel and S. C. Khoo. Semantics-directed generation of a Prolog compiler. In J. Maluszyński and M. Wirsing, editors, *3^{th} Symposium on Programming Languages Implementation, and Logic Programming, Passau, Germany, Aug 1991*, LNCS 528, pages 135–146. Springer-Verlag, 1991.

[Con90a] C. Consel. Bindings time analysis for higher order untyped functional languages. In *ACM Conference on on Lisp and Functional Programming*, pages 264–272, 1990.

[Con90b] C. Consel. *The Schism manual v1.0.* Yale University, New Haven, Connecticut, USA, New Haven, Connecticut, 1990.

[Con91] C. Consel. Evaluation partielle : principes, pratique et application. Document d'habilitation, IRISA Rennes, Octobre 1991.

[Cou79] B. Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *J. Theoretical Computer Science*, 25:95–169, 1983.

[Cou86] B. Courcelle. Equivalences and transformations of regular systems – application to recursive program schemes and grammars. *J. Theoretical Computer Science*, 42:1–122, 1986.

[Cou90a] B. Courcelle. Recursive applicative program schemes. In Leeuwen [Lee90], pages 459–492.

[Cou90b] Bruno Courcelle. Graph rewriting. In Leeuwen [Lee90], pages 193–242.

[Dan91] O. Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37:315–322, 1991.

[Dau89] Max Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In *Proc. 3^{rd} International Conference on Rewriting Techniques and Applications, Chapel Hill, NC*, LNCS 355, pages 109–120. Springer-Verlag, 1989.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *J. Theoretical Computer Science*, 17:279–301, 1982.

[Der87] Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.

[Des83] Thierry Despeyroux. *Spécifications sémantiques dans le système Mentor*. Thèse de doctorat, Université de Paris XI, Orsay, France, Octobre 1983.

[Des84] Thierry Despeyroux. Executable specification of static semantics. In *Proc. Symposium on Semantics of Data Types, Sophia Antipolis*, LNCS 173. Springer-Verlag, 1984.

[Des88] Thierry Despeyroux. TYPOL: a formalism to implement natural semantics. Rapport Technique RT-94, INRIA, France, Mars 1988.

[Dev90] Philippe Devienne. Weighted graphs: a tool for studying the halting problem and time complexity in term rewriting systems and logic programming. *J. Theoretical Computer Science*, 75:157–215, 1990.

[DG82] Roland Dubois and Dominique Girod. *Les microprocesseurs 16 bits à la loupe*. Eyrolles, 1982.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Leeuwen [Lee90], pages 243–320.

[DLR93] P. Devienne, P. Lebègue, and J.-C. Routier. Halting problem of one binary Horn clause is undecidable. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Proc. 10 th Symposium on Theoretical Aspects of Computer Science, Würzburg, Germany, Feb 1993*, LNCS 665, pages 48–57. Springer-Verlag, 1993.

[DS76] P. Downey and R. Sethi. Correct computation rules for recursive languages. *SIAM J. of Computing*, 5, 1976.

[EBF88] A.P. Ershov, D. Bjørner, and Y. Futamura, editors. *Partial evaluation and mixed computation: selected papers*, volume 6 of *New Generation Computing*. Ohmsha ltd. and Springer-Verlag, 1988.

[Ede85] E. Eder. Properties of substitutions and unifications. *J. Symbolic Computation*, 1:31–46, 1985.

[Ers78] Andrei P. Ershov. On the essence of compilation. In E.J. Neuhlod, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[Ers88] Andrei P. Ershov. Opening key-note speech. In Bjørner et al. [BEJ88], pages xxiii–xxix.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In Bjørner et al. [BEJ88], pages 133–151.

[FNT91] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *J. Theoretical Computer Science*, 90:61–79, 1991.

[FOF88] H. Fujita, A. Okumura, and K. Furukawa. Partial evaluation of GHC programs based on the Ur-set with constraints. In R.A. Kowalski and K.A. Bowen, editors, *Proc. 5 th International Conference on Logic Programming, Seattle, USA, August 1988*, pages 924–941. MIT Press, 1988.

[FSZ89] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda–Upsilon–Omega: The 1989 Cookbook. Rapport de Recherche 1073, INRIA, France, Août 1989. The system... 116 pages.

[FSZ91] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average–case analysis of algorithms. *J. Theoretical Computer Science, Series A*, 79(1):37–109, Février 1991. The theory...

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[Fut88] Y. Futamura. Program evaluation and generalized partial computation. In *Proc. of the international Conference on fifth Generation Computer Systems*, pages 685–692. ICOT, 1988.

[GK92] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proc. Conference on Programming Languages, Design and Implementations, San Francisco, June 1992*, SIGPLAN Notices. ACM Press, 1992.

[Gri91] S. Grigorieff. *Décidabilité et complexité*. Notes de cours du GRECO de programmation, 1991.

[GS90] C.A. Gunter and D. Scott. Semantics domains. In Leeuwen [Lee90], pages 633–674.

[GS91] P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations of logic programs. In Lassez and Plotkin [LP91], pages 565–583.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebra. *J. Association for Computing Machinery*, 24(1):68–95, Janvier 1977.

[Gue81] Irène Guessarian. *Algebraic semantics*. LNCS 99. Springer-Verlag, 1981.

[Gur91] Douglas J. Gurr. *Semantics frameworks for complexity*. PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, Scotland, Janvier 1991. CST-72-91. Aussi publié comme ECS-LFCS-91-30.

[Gus92] Jens Gustedt. *Algorithmic aspects of ordered structures*. PhD thesis, Technische Universität Berlin, 1992.

[Han91] John Hannan. Staging transformations for abstract machines. In PEPM'91 [PEP91], pages 130–141.

[Has88] Laurent Hascoët. Partial evaluation with inference rules. In PEMC'87 [EBF88], pages 187–209.

[Hee86] Jan Heering. Partial evaluation and ω-completeness of algebraic specifications. *J. Theoretical Computer Science*, 43:149–167, 1986.

[HG91] C. K. Holst and C. K. Gomard. Partial evaluation is fuller lazyness. In PEPM'91 [PEP91], pages 223–233.

[HH85] H. Herrlich and M. Husek. Galois connections. In A. Melton, editor, *Proc. Mathematical Foundations of Programming Semantics, Manhattan, KS, April 1985*, LNCS 239, pages 122–134. Springer-Verlag, 1985.

[HHP91] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. Technical Report ECS-LFCS-91-162, LFCS, Department of Computer Science, University of Edinburgh, Scotland, Juin 1991.

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, 2(3):326–336, 1952.

[HL91] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In Lassez and Plotkin [LP91], chapter 11, 12, pages 395–443.

[HM90] John Hannan and Dave Miller. From operational semantics to abstract machines: preliminary results. In M. Wand, editor, *Proc. ACM Conference on Lisp and Functional Programming, Nice, France, June 1990*, pages 323–332. ACM Press, Juin 1990.

[HM92] S. Harnett and M. Montenyohl. Toward efficient compilation of a dynamic object-oriented language. In PEPM'92 [PEP92], pages 82–89.

[HMT89] R. Harper, R. Milner, and M. Tofte. The definition of Standard ML Version 3. Technical Report ECS-LFCS-89-81, LFCS, Department of Computer Science, University of Edinburgh, Scotland, Mai 1989. Voir aussi [MTH90].

[HO80] G. Huet and D.C. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal language theory: perspective and open problems*, pages 349–405. Academic Press, 1980.

[HS91] T. J. Hickey and D. A. Smith. Toward the partial evaluation of CLP languages. In PEPM'91 [PEP91], pages 43–51.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[Hue76] G. Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse d'État, Université de Paris VII, France, Septembre 1976.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Association for Computing Machinery*, 27(4):797–821, Octobre 1980.

[Hue88] Gerard Huet. A uniform approach to type theory. Rapport de Recherche RR-795, INRIA, France, Février 1988.

[Hug91] J. Hugues, editor. *Proc. 5^{th} Conference on Functional Programming Languages and Computer Architecture, MA, USA, Aug 1991*, LNCS 523. Springer-Verlag, 1991.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[JM84] N.D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Int. Symposium on Logic Programming, Atlantic City, New Jersey, 02/1984*, pages 281–288, Février 1984.

[Jon88] N.D. Jones. Challenging problems in partial evaluation and mixed computation. In Bjørner et al. [BEJ88], pages 1–14.

[Jør92] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 258–268, 1992.

[JR92] Ian "Jake" Jacobs and Laurence Rideau. A Centaur tutorial. Rapport Technique RT-140, INRIA, France, Août 1992.

[JSS85] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, LNCS 202, pages 124–140. Springer-Verlag, 1985.

[JSS89] N. D. Jones, P. Sestoft, and H. Søndergaard. A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[Kah87] G. Kahn. Natural semantics. In *Proceedings of STACS, march 1987*, LNCS 247. Springer-Verlag, 1987. Aussi publié comme Rapport de Recherche INRIA RR-601, Février 1987.

[KB70] D. E. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[KK88a] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in fold/unfold logic program transformation. In *Proc. of the international Conference on fifth Generation Computer Systems*, pages 413–421. ICOT, 1988.

[KK88b] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in fold/unfold logic program transformation (ii). Rapport Technique TR-403, ICOT, 1988.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. Princeton, 1952.

[Kom82] H.J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. In *Proc. 9 th ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 255–267. ACM Press, 1982.

[Kor85a] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Kor85b] R.E. Korf. Iterative-deepening-A*: an optimal admissible tree search. In A. Joshi, editor, *Proc. 9 th International Joint Conference on Artificial Intelligence, Los Angeles, California, August 1985*, volume 2, pages 1034–1036. Morgan Kaufmann Publishers, Inc., Août 1985.

[Kot80] L. Kott. *Des substitutions dans les systèmes d'équations algébriques sur le magma : application aux transformations de programmes et à leur correction*. Thèse d'État, Université de Paris VII, France, 1980.

[Kot85] L. Kott. Unfold/fold program transformations. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12, pages 411–434. Cambridge University Press, 1985. Aussi publié comme Rapport de Recherche INRIA RR-155, Août 1982.

[Kru60] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and Vázsonyi's conjecture. *J. Transaction of the American Math. Society*, 95:210–225, 1960.

[KS91] S. C. Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In PEPM'91 [PEP91], pages 211–222.

[Lau91a] J. Launchbury. *Projection factorisation in partial evaluation*. Cambridge University Press, 1991.

[Lau91b] John Launchbury. A strongly-typed self-applicable partial evaluation. In Hugues [Hug91], pages 145–164.

[Lee90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B. Formal models and semantics. Elsevier, 1990.

[Lév80] Jean-Jacques Lévy. Optimal reduction in the λ-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry, Essays on combinatory logic, lambda-calculus and formalism*, pages 159–191. Academic Press, 1980.

[Liv78] C. Livercy. *Théorie des programmes*. Dunod, 1978.

[Llo87] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, second edition, 1987.

[LM88] Daniel Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 20(2):248–266, Avril 1988.

[LP91] J.-L. Lassez and G. Plotkin, editors. *Computational Logic – Essays in honor of Alan Robinson*. MIT Press, 1991.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Logic Programming*, 11:217–242, 1991.

[Mal92] K. Malmkjær. Predicting properties of residual programs. In PEPM'92 [PEP92], pages 8–13.

[Mas87] H. Massalin. Superoptimizer: A look at the smallest program. In *Proc. 2 nd Int. Conference on Architecture Support for Programming Languages and Operating Systems, Palo Alto, October 1987*, pages 122–126, 1987.

[Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In PEPM'91 [PEP91], pages 94–105.

[Mog86] T. Æ. Mogensen. The application of partial evaluation to ray tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[Mog89a] T. Æ. Mogensen. *Binding time aspects of partial evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, Mars 1989.

[Mog89b] Eugenio Moggi. Computational lambda-calculs and monads. In *Proc. 4 th Conference on Logic in Computer Science*, 1989. Aussi publié comme Rapport de Recherche LFCS, ECS-LFCS-88-66, 1988.

[Mog90] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, Department of Computer Science, University of Edinburgh, Scotland, 1990.

[Mos90] P.D. Mosses. Denotational semantics. In Leeuwen [Lee90], pages 576–631.

[MS91] Y. Moscowitz and E. Shapiro. Lexical logic programs. In K. Furukawa, editor, *Proc. 8 th International Conference on Logic Programming, Paris, France, June 1991*, pages 349–363, 1991.

[MSS85] A. Melton, D. A. Schmidt, and G. E. Strecker. Galois connections and computer science applications. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category theory and computer programming*, LNCS 240, pages 299–312. Springer-Verlag, Septembre 1985.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990. Voir aussi [HMT89].

[Nie88] F. Nielson. Comparing partial evaluators. In Bjørner et al. [BEJ88], pages 349–384.

[NW63] C. ST. J. A. Nash-William. On well-quasi-ordering finite trees. *Proceedings of the Cambridge philosophical society*, 59:833–835, 1963.

[Pau91] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[PEP91] *Proc. Symposium on Partial Evaluation and semantics-based Program Manipulation, New Haven, June 1991*, volume 26(9) of *SIGPLAN Notices*. ACM Press, Septembre 1991.

[PEP92] *Proc. ACM SIGPLAN Workshop on Partial Evaluation and semantics-based Program Manipulation, San Francisco, June 1992*, Rapport Technique YALEU/DCS/RR-909, New Haven, Connecticut, 1992. Yale University, New Haven, Connecticut, USA.

[PEP93] *Proc. Symposium on Partial Evaluation and semantics-based Program Manipulation, Copenhagen, Denmark, June 1993*, SIGPLAN Notices. ACM Press, 1993.

[Pla86] D.A. Plaisted. A simple non-termination test for the Knuth-Bendix method. In Jörg Siekmann, editor, *Proc. 8th International Conference on Automated Deduction, Oxford, England*, LNCS 230, pages 79–88. Springer-Verlag, 1986.

[Plo70] G. D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. American Elsevier, 1970.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Septembre 1981.

[PP91a] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for prolog. In PEPM'91 [PEP91], pages 274–284.

[PP91b] M. Proietti and A. Pettorossi. Unfolding - definition - folding, in this order, for avoiding unnecessary variables in logic programs. In J. Maluszyński and M. Wirsing, editors, *3th Symposium on Programming Languages Implementation, and Logic Programming, Passau, Germany, Aug 1991*, LNCS 528, pages 347–358. Springer-Verlag, 1991.

[Pre92] S. Prestwich. *The PADDY user guide*. ECRC, München, Germany, Janvier 1992.

[QG92] C. Queinnec and J. M. Geffroy. Partial evaluation applied to pattern matching with intelligent backtracking. Rapport Technique LIX-RR-92-14, LIX, Ecole polytechnique, Palaiseau, France, 1992.

[Rao89] Jean-Claude Raoult. Mémento sur les ensembles ordonnés. Publication interne 503, IRISA, Rennes, Novembre 1989.

[RDO77] E. Ramis, C. Deschamps, and J. Odoux. *Cours de Mathématiques spéciales — Séries et équations différentielles*, volume 2. Masson, 1977.

[Rom88] A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Bjørner et al. [BEJ88], pages 445–463.

[Rom91] A. Romanenko. Inversion and metacomputation. In PEPM'91 [PEP91], pages 12–21.

[Ros73] B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *J. Association for Computing Machinery*, 20(1):160–187, Janvier 1973.

[Ros89] Mads Rosendahl. Automatic complexity analysis. In *Proc. 4th Conference on Functional Programming Languages and Computer Architecture, London, Sept 1989*, pages 144–156. ACM Press, 1989.

[RV80] J.-C. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *J. Association for Computing Machinery*, 27(4):772–796, Octobre 1980.

[Sah91a] D. Sahlin. *An automatic partial evaluator for full Prolog*. PhD thesis, Swedish Institute of Computer Science, Kista, Sweden, Mars 1991.

[Sah91b] D. Sahlin. *Mixtus V0.3.3*. Kista, Sweden, Mai 1991. Online reference manual page.

[San90] D. Sands. Complexity analysis for a lazy higher order language. In *Proc. 3rd European Symposium on Programming*, LNCS 432. Springer-Verlag, Mai 1990.

[San91] D. Sands. Time analysis, cost equivalence and program refinement. In *Proc. 11 th Conference on on Foundations of Software Technology and Theoretical Computer Science*, LNCS 560, pages 25–39. Springer-Verlag, Décembre 1991.

[San93] D. Sands. A naïve time analysis and its theory of cost equivalence. TOPPS report D-173, DIKU, University of Copenhagen, Denmark, 1993. Submitted for publication.

[Sav82] W. J. Savitch. *Abstract machines and grammars*. Little, Brown & Company, 1982.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[Smi91] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming. In PEPM'91 [PEP91], pages 62–71.

[SSD91] D. Sherman, R. Strandh, and I. Durand. Optimisation of equational programs using partial evaluation. In PEPM'91 [PEP91], pages 72–82.

[ST85] M.E. Stickel and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In A. Joshi, editor, *Proc. 9 th International Joint Conference on Artificial Intelligence, Los Angeles, California, August 1985*, volume 2, pages 1073–1075. Morgan Kaufmann Publishers, Inc., Août 1985.

[Sti88] Jonathan Stillman. *Computational problems in equational theorem proving*. PhD thesis, State University of New York at Albany, 1988.

[Sto88] Allen Stoughton. *Fully abstract models of programming languages*. Research Notes in Theoretical Computer Science. Pitman Publ., 1988. Voir aussi Ph.D. Thesis, Technical Report CST-40-86, CS Dept, University of Edinburgh, 1986.

[Sun91] R. S. Sundaresh. Building incremental programs using partial evaluation. In PEPM'91 [PEP91], pages 83–93.

[SZ88] P. Sestoft and A. V. Zamulin. Literature list for partial evaluation and mixed computation. In Bjørner et al. [BEJ88], pages 589–622.

[Tak91] A. Takano. Generalized partial computation for a lazy functional language. In PEPM'91 [PEP91], pages 1–11.

[Toy88] Y. Toyama. Commutativity of term rewriting systems. In K. Fuchi and L. Kott, editors, *Programming of future generation computers II*, pages 393–407. Elsevier (North-Holland), 1988.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proc. 2 th International Conference on Logic Programming, Uppsala, Sweden, 1984*, pages 127–138, 1984.

[Tur85] V. F. Turchin. Program transformation by supercompilation. In H. Ganzinger and N.D. Jones, editors, *Proc. Programs as Data Objects, Copenhagen, Denmark, 1985*, LNCS 217, pages 257–281. Springer-Verlag, 1985.

[Tur86] V. F. Turchin. The concept of supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, Juillet 1986.

[Tur88] V. F. Turchin. The algorithm of generalization in the supercompiler. In Bjørner et al. [BEJ88], pages 531–549.

[Var94] Moshe Y. Vardi. Nontraditional applications of automata theory. In M. Hagiya and J.C. Mitchell, editors, *Symposium on Theoretical Aspects of Computer Science, Sendai, Japan, April 1994*, LNCS 789, pages 575–597. Springer-Verlag, 1994.

[Vui74] J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. *J. Computatiuonal System Sciences*, 9:332–354, 1974.

[Wal88] P. Walder. Deforestation: transforming programs to eliminate trees. In H. Ganzinger, editor, *Proc. 2 nd European Symposium on Programming, Nancy, France, Mar 1988*, LNCS 300, pages 344–358. Springer-Verlag, 1988.

[War83] David H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, Ca, Octobre 1983.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In Hugues [Hug91], pages 165–191.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, Septembre 1975.

[Wir90] M. Wirsing. Algebraic specifications. In Leeuwen [Lee90], pages 675–788.

Index des Figures

Index des Tableaux

Index des Définitions

Index des Propositions, Lemmes et Théorèmes

Index des Symboles et des Termes

En règle générale, les numéros de pages qui figurent en *italique* renvoient à la définition ou à un emploi majeur. Les termes et les notations données à l'annexe N ne figurent pas dans cet index.