HAL
open science

Caractérisation impérative des algorithmes séquentiels en temps quelconque, primitif récursif ou polynomial

Yoann Marquer

To cite this version:

Yoann Marquer. Caractérisation impérative des algorithmes séquentiels en temps quelconque, primitif récursif ou polynomial. Informatique et langage [cs.CL]. Université Paris-Est, 2015. Français. NNT : 2015PESC1121 . tel-01280467

HAL Id: tel-01280467

https://theses.hal.science/tel-01280467

Submitted on 29 Feb 2016

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

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

UNIVERSITÉ PARIS-EST

lacl

École doctorale MSTIC

Doctorat de recherche en informatique

Caractérisation impérative des algorithmes séquentiels en temps quelconque, primitif récursif ou polynomial

Auteur
Yoann Marquer

Directeur de thèse
Pierre Valarcher

Thèse soutenue le vendredi 9 octobre 2015

Rapporteurs

Patrick Baillot, *Directeur de recherche au CNRS*

Gilles Dowek, *Directeur de recherche à INRIA*

Daniel Leivant, *Professeur à l'Université de l'Indiana à Bloomington*

Membres du Jury

Patrick Baillot, *Directeur de recherche au CNRS*

Patrick Cégielski, *Professeur à l'Université Paris-Est Créteil*

Gilles Dowek, *Directeur de recherche à INRIA*

Jean-Yves Marion, *Professeur à l'Université de Lorraine*

Pierre Valarcher, *Professeur à l'Université Paris-Est Créteil*

Remerciements

Avant toute chose je souhaiterais remercier Solenn Thomas, ma fiancée, qui a toujours été là pour moi tout en proposant un regard différent mais souvent juste, ainsi que Pierre Valarcher, mon directeur de thèse, qui a réussi le numéro d'équilibriste de me guider tout en me laissant trouver ma voie. En m'aidant sans complaisance vous m'avez beaucoup fait progresser, et ce manuscrit n'aurait pas existé sans vous.

Je remercie également ma famille et mes amis qui ont supporté mes longues réflexions techniques ou philosophiques durant toutes ces années, ainsi que mes collègues pour leurs retours et leur aide notamment pour mes enseignements. Je remercie en particulier mes collègues doctorants à la fois pour leur soutien technique mais aussi pour nos discussions passionnantes.

Le style bibliographique de ce manuscrit est emprunté à la Société Mathématique de France, les notes de bas de page utilisent le package footnotebackref, les sommaires en début de chapitre sont dus au package minitoc, et l'aspect des dialogues au package dialogue. En passant, je remercie les membres du forum MathemaTex.net pour leurs réponses.

Cette thèse est partiellement financée par l'ANR 12 BS02 007 01.

Guide d'utilisation

Les liens bleus permettent de naviguer dans le manuscrit :
- Cliquer sur un numéro de page mènera à la page correspondante.
- Cliquer dans la table des matières mènera au début du chapitre ou de la section.
- Cliquer sur un titre de chapitre ramènera à la table des matières générale.
- Cliquer sur un titre de section ou de sous-section ramènera au sommaire du chapitre correspondant.

Les liens cyans permettent d'accéder à l'ouvrage correspondant dans la bibliographie, où les numéros des pages où cet ouvrage est cité permettent de revenir.
Les liens magenta donnent sur des élément extérieurs au manuscrit, comme une adresse mail ou un article de Wikipédia.

Discussion préliminaire

Un dialogue est plus long, à lire comme à écrire, qu'un simple résumé. C'est probablement pour cela que c'est une forme peu utilisée en sciences, où pourtant la critique par les pairs mais aussi l'autocritique est encouragée : la démarche scientifique a ceci d'original que nous cherchons surtout à prouver que nous avons tort, jusqu'à ce que notre édifice mental semble assez solide pour être examiné par d'autres.

Mais quand il s'agit de présenter notre travail il nous arrive de taire des hypothèses ou des complications, de laisser des « détails sous le tapis », parfois inconsciemment. C'est moins le cas dans un travail exhaustif comme celui de la thèse, mais cette tendance peut être évitée tout simplement en donnant la parole à un « autre », qui ne se réduit pas forcément à un « avocat du diable » mais peut être aussi le collègue ou l'ami soucieux de comprendre notre démarche, afin de s'instruire ou de nous éviter de commettre une erreur :

> *Je suis de ceux qui ont plaisir à être réfutés, s'ils disent quelque chose de faux, et qui ont plaisir aussi à réfuter les autres, quand ils avancent quelque chose d'inexact, mais qui n'aiment pas moins à être réfutés qu'à réfuter. Je tiens en effet qu'il y a plus à gagner à être réfuté, parce qu'il est bien plus avantageux d'être soi-même délivré du plus grand des maux que d'en délivrer autrui.* — (Socrate [1], *Gorgias*)

La dialogue, en présentant une pluralité des points de vue et en soulevant naturellement des questions légitimes, montre non seulement le résultat de la recherche mais aussi son processus, trop ignoré du grand public qui voit souvent la science comme quelque chose à croire plutôt que quelque chose à connaître. Le dialogue a également un intérêt pédagogique de par son dynamisme et son humanité par rapport à un simple manifeste [2].

Pour cela nous suivrons l'exemple de Gurevich [3] en utilisant un dialogue entre l'auteur et un collègue fictif appelé Quisani à deux endroits de ce manuscrit : nous discuterons ici des questions ayant motivé mon travail de thèse, et dans la conclusion p.105 nous le critiquerons en examinant les « détails laissés sous le tapis ».

QUISANI : « Caractérisation impérative des algorithmes séquentiels en temps quelconque, primitif récursif ou polynomial » : voilà un titre certes impressionnant par son jargon, mais qui me laisse assez perplexe... De quoi parles-tu exactement ?

AUTEUR : [*réfléchit un moment*] Je pense que dans un premier temps il faudrait nous mettre d'accord sur la notion d'algorithme.

1. ↑ Rapporté par Platon et traduit par Émile Chambry.
2. ↑ Je recommande par exemple l'introduction aux nombres surréels de Donald Knuth [Knu74], dont une traduction en français a été faite par Daniel et Hélène Loeb.
3. ↑ Par exemple dans [BG04] où il discute de l'utilisation des ensembles comme briques de base des mathématiques.

Q : Je pensais que cette notion-là était claire... Prenons par exemple la définition de Knuth, à savoir qu'un algorithme doit :

1. **être fini** : « Un algorithme doit toujours se terminer après un nombre fini d'étapes. »
2. **être bien défini** : « Chaque étape d'un algorithme doit être définie précisément, les actions à transposer doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas. »
3. **avoir des entrées** : « ... des quantités qui lui sont données avant qu'un algorithme ne commence. Ces entrées sont prises dans un ensemble d'objets spécifié. »
4. **avoir des sorties** : « ... des quantités ayant une relation spécifiées avec les entrées. »
5. **être effectif** : « ... toutes les opérations que l'algorithme doit accomplir doivent être suffisamment basiques pour pouvoir être en principe réalisées dans une durée finie par un homme utilisant un papier et un crayon. »

A : Nous pourrions discuter du fait que certains algorithmes peuvent être légitimement infinis, comme le crible d'Eratosthène qui pourrait donner la suite ininterrompue des nombres premiers, mais partons de cette esquisse dans un premier temps. Je parle d'esquisse bien sûr parce que bien qu'intuitive ta collection de critères n'est pas une définition formelle.

Q : Je te trouve un peu malhonnête... comme c'est toi qui parle d'algorithmes ça devrait être à toi de m'expliquer ce qu'ils sont. Mais puisque je le sais je vais répondre à ta question. Ces critères correspondent à la notion intuitive de « méthode effective de calcul », qui a été décrite formellement par différents modèles de calcul comme les fonctions récursives de Gödel et Herbrand, le lambda-calcul de Church et Kleene ou les machines de Turing.

A : Excuse-moi, je ne voulais pas te froisser, juste être sûr que nous parlions de la même chose. D'ailleurs, tu ne m'as pas donné un modèle mais trois. Je me doute bien de pourquoi tu le fais, mais allons jusqu'au bout si tu le veux bien. Est-ce trois fois la même définition ?

Q : Pas de problème. Oui, en effet Kleene et Rosser ont montré que les fonctions récursives coïncidaient avec le lambda-calcul, et Turing a montré que ses machines coïncidaient également avec le lambda-calcul. C'est d'ailleurs parce que ces trois tentatives de formaliser la notion de « méthode effective de calcul » coïncident que nous croyons que cette notion intuitive a bien été capturée par ces formalismes : c'est la thèse de Church.

A : Donc, si je comprends bien, pour toi la question de savoir ce qu'est un algorithme est réglée depuis les années 1930 : un algorithme est un calcul effectué par une machine de Turing. C'est bien ça ?

Q : Je dirais même plus que ce n'est pas seulement mon avis, mais le consensus actuel. D'où mon étonnement quand tu as demandé à ce que nous nous entendions sur une définition si vieille [Tur37] et si communément acceptée. À moins bien sûr que tu ne me réserves une surprise ?

A : Il y a un peu de ça... Connais-tu l'article [Col91] de Colson ?

Q : Pas du tout. Tu pourrais me faire un court résumé ?

A : Il s'intéresse aux fonctions récursives primitives [1] mais d'un point de vue intentionnel et non extensionnel : pas ce qui peut être calculé, mais comment le calcul est fait. Pour cela il utilise comme entrées des entiers normaux de la forme $S^k 0$ mais aussi des « entiers paresseux » de la forme $S^k \perp$, signifiant que nous savons que l'entrée commence par k symboles S. Si le calcul arrive sur un $\perp$ il s'arrête alors à cause du manque d'information.

Q : Attends, je reformule... L'entrée est donnée au compte-goutte : un fragment d'information est révélé, permettant de commencer le calcul, puis une fois cette partie du calcul achevée un autre fragment est révélé, etc. J'ai saisi l'idée ?

A : En gros. Ce qui est surprenant, c'est qu'en utilisant ce modèle Colson arrive à démontrer que les fonctions récursives primitives ne permettent pas de calculer l'inf de m et n en un temps $O(\mathit{inf}(m,n))$. J'y reviendrai notamment à la p.44.

Q : Mais l'*inf* est pourtant une fonction primitive récursive !

A : Oui, il peut être calculé. En temps $O(m)$ ou $O(n)$ par exemple, mais pas en $O(\mathit{inf}(m,n))$ étapes : on ne peut pas faire décroître alternativement les deux arguments. Le schéma de récursion s'obstine [2] à la place sur un des deux, ce qui nous prive d'un bon algorithme.

Q : [*marmonne*] Je ne suis pas convaincu. C'est surement dû à une implémentation particulière des fonctions récursives primitives, ou plutôt à un schéma de récursion trop limité. On doit forcément pouvoir coder les arguments de façon à pouvoir implémenter le bon algorithme...

A : Doucement, une chose à la fois !

1. D'une part l'implémentation utilisée est celle des combinateurs de Kleene, qu'on peut trouver dans des ouvrages comme [CL03b], donc elle semble assez canonique.

2. D'autre part, effectivement le schéma de récursion importe. Par exemple nous savons [Gla67] qu'enlever la variable de récursion dans le schéma habituel permet de calculer les mêmes fonctions, mais le prédécesseur ne se calcule plus en une seule étape : il est même assez difficile à calculer. Mon directeur de thèse, Pierre Valarcher, a d'ailleurs montré dans [MV09] qu'en utilisant un schéma de récursion plus large l'*inf* pouvait bien être calculé en $O(\mathit{inf}(m,n))$ étapes. De façon plus générale ce schéma permet de calculer une vaste classe d'algorithmes appelée `APRA` (Arithmetical Primitive Recursive Algorithms).

3. Enfin, les codages ne sont jamais gratuits. Par exemple le codage connu $\alpha(m,n) = \frac{1}{2}(m+n+1)(m+n)+n$ permettant de montrer que $\mathbb{N}^2$ est en bijection avec $\mathbb{N}$ est très loin de se faire en temps constant...

Q : Je continuerai à y réfléchir [3]... mais je doute que le problème relevé par Colson puisse s'appliquer également à la définition sur laquelle nous nous sommes entendus pour les algorithmes : à savoir, un calcul effectué par une machine de Turing.

A : Utilisant combien de rubans ?

1. ↑ Passé ce dialogue, nous aurons malheureusement tendance à utiliser l'anglicisme de « fonction primitive récursive » au lieu de « fonction récursive primitive ».

2. ↑ David a étendu dans [Dav01] le théorème d'ultime obstination de Colson à la récursion mutuelle, et y propose une approche similaire pour des structures de données quelconques.

3. ↑ A ce propos, je tiens à remercier Konstantin Verchinine de l'IUT de Sénart-Fontainebleau pour ses critiques constructives.

Q : Qu'importe, puisqu'il a été montré que ces machines étaient équivalentes.

A : Oh, je ne prétends pas qu'avec davantage de rubans il soit possible de calculer davantage de fonctions, je ne parlais que d'algorithmes. Par exemple il a été montré [BBD+04] que la reconnaissance d'un palindrome nécessite au moins $O(n^2/log(n))$ étapes sur une machine de Turing avec une seule bande, alors que nous savons que $O(n)$ étapes suffisent avec deux bandes. C'est bien la même fonction, mais comme les temps d'exécution sont différents ce ne sont pas les mêmes algorithmes.

Q : J'en conviens, mais cela contredit notre définition précédente de ce qu'est un algorithme. D'ailleurs j'ai remarqué que tu distinguais « fonction » et « algorithme » dans ton discours. Je trouve dommage d'avoir à donner de nouveaux sens à ces mots que l'usage a consacré, mais vu tes exemples j'en comprends la nécessité. Alors, quel sens leur donnes-tu ?

A : Je parle de « fonction » au sens mathématique, c'est à dire l'objet qui à des entrées associe une sortie. Quand je dis qu'un modèle calcule une fonction je dis qu'il y existe un programme tel qu'à partir d'un état initial donné il est possible d'obtenir l'état final correspondant. La notion d'« algorithme » devrait inclure le temps d'exécution, c'est à dire le nombre d'étapes, mais pas seulement.

Q : Parles-tu de la trace de l'exécution ?

A : Je dis que toutes les informations contenues dans l'exécution, c'est à dire le nombre d'étapes, les valeurs successives des variables dans la mémoire, etc. devraient probablement être considérées comme faisant partie de l'algorithme. En un sens, l'algorithme serait l'exécution elle-même.

Q : Mais que fais-tu de la thèse de Church ?

A : Elle s'appuie sur le fait que les fonctions récursives, le lambda-calcul et les machines de Turing calculent le même ensemble de fonctions pour affirmer que ce sont bien les « fonctions calculables ». Mais c'est un résultat sur les fonctions et non les algorithmes : il nous faut une nouvelle thèse pour les algorithmes. Ce que Colson a démontré, c'est que les combinateurs de Kleene permettaient bien de calculer toutes les fonctions récursives primitives, mais pas de tous les moyens possibles : ce modèle est incomplet algorithmiquement parlant.

Q : Existe-t-il des langages de programmation qui soient algorithmiquement complets ?

A : C'est mot pour mot la question que m'a posée Pierre Valarcher lors de notre première rencontre, et qui a motivé mes travaux. Bien sûr, encore une fois, la question est de savoir ce que nous entendons exactement par algorithme.

Q : Ah, je comprends mieux pourquoi tu as pris tant de précautions autour de ce mot. Mais si le but est d'avoir une thèse similaire à celle de Church, il suffit de prendre des formalisations différentes des algorithmes, de voir celles qui coïncident, et ensuite d'en déduire que la notion intuitive d'algorithme correspond aux objets obtenus par ces formalisations. Certains ont bien dû s'intéresser au problème, non ?

A : Hélas, ce qui a bien marché pour les fonctions ne semble pas encore clair pour les algorithmes. Il y a bien eu plusieurs tentatives, je peux t'en détailler trois si tu veux, mais elles ne semblent pas coïncider.

Q : Je veux bien, mais ne sois pas trop exhaustif car si nous mettons autant de temps à cerner les autres mots de ton titre cette discussion me semblera interminable.

A : Je vais essayer d'aller à l'essentiel alors, du plus particulier au plus général :

1. Yanofsky s'est focalisé dans [Yan11] uniquement sur les fonctions primitives récursives en prenant au sérieux l'idée que plusieurs programmes correspondent au même algorithme et plusieurs algorithmes à la même fonction, la question étant alors de chercher des relations d'équivalences appropriées. Dans [Yan10] il fait même une « théorie de Galois » des algorithmes en cherchant les relations les plus élégantes possibles du point de vue des catégories. Toutefois ces choix restent assez subjectifs [1].

2. Gurevich a formalisé dans [Gur00] les algorithmes séquentiels avec ses Abstract State Machines (ASMs), qu'on peut voir comme un ensemble de règles de la forme `if` *condition* `then` *actions* assez similaires à [Ars91]. Elles ont été implémentées avec le langage de programmation `AsmL` [2]. Moschovakis a critiqué cette démarche se restreignant aux algorithmes implémentables de façon mécanique.

3. Moschovakis [3] a formalisé dans [Mos01] la notion d'algorithme comme étant un système d'équations récursives (similaire à notre exemple du *pgcd* p.45). Les ASMs, comme tout système à transition d'états (ou « implémentation » selon Moschovakis), devraient pouvoir y être décrits, mais pas seulement. Gurevich a répondu dans [BG02] que les systèmes d'équations de Mochovakis étaient davantage des spécifications que des algorithmes, et que des machines plus générales pouvaient implémenter les algorithmes non-mécaniques de Moschovakis.

Q : Effectivement, il ne semble pas y avoir de consensus. Mais vu comment tu décris les modèles de Gurevich et Moschovakis j'ai davantage l'impression d'une opposition entre paradigmes de programmation, disons impératif contre fonctionnel, que d'une différence réelle entre les objets eux-mêmes.

A : C'est possible, mais je n'en sais pas assez pour te répondre [4]. J'ai donc dû faire un choix, et la présentation de Gurevich m'a semblé plus convaincante.

Q : Qu'est-ce qui a fait pencher la balance ?

A : Il faut dire qu'à l'époque j'étais assez d'accord sur le fait que la récursion ressemblait davantage à de la spécification qu'à de la programmation, dans la mesure où en plus du programme il fallait généralement préciser la stratégie d'évaluation, mais je suis devenu plus mitigé sur la question. Toutefois ce qui m'a vraiment convaincu est la présentation axiomatique de Gurevich.

Q : Il a posé des axiomes pour ses ASMs ?

A : Non, il a exprimé trois postulats que devraient vérifier les algorithmes séquentiels : c'est la thèse de Gurevich. Puis ensuite il a défini ses ASMs et montré que les deux notions coïncidaient bien. C'est parce que j'ai trouvé ses trois postulats convaincants que j'ai accepté les ASMs comme modèle pour les algorithmes séquentiels. Je les détaille longuement à la section 1.2 p.22, mais en voici une version résumée :

1. ↑ Voir par exemple [BDG09] pour une discussion plus complète.
2. ↑ Voir [CG10] pour une comparaison.
3. ↑ Il a également prouvé dans [Mos03] un résultat similaire à l'*inf* de Colson, mais pour le *pgcd*.
4. ↑ Nous abordons toutefois à nouveau la question p.110.

1. **Temps séquentiel** : Le calcul est effectué étape par étape.
2. **États abstraits** : Les états sont représentables, à isomorphisme près.
3. **Exploration bornée** : Une quantité bornée de lectures ou d'écritures est faite à chaque étape.

Q : Effectivement, cela paraît assez naturel. Mais quelle nuance fais-tu quand tu parles d'« algorithme séquentiel » par rapport à « algorithme » ?

A : Dans mon manuscrit je ne traiterai que des algorithmes vérifiant ces trois postulats, appelés par Gurevich les algorithmes séquentiels, et qui incluent la plupart des algorithmes usuels, comme par exemple l'algorithme d'Euclide. Mais cette formalisation n'inclut pas les algorithmes parallèles, distribués, non-déterministes, quantiques, génétiques, etc. Il y a eu des tentatives, par exemple celle de Gurevich [BG08] pour formaliser les algorithmes parallèles, mais rien d'aussi convaincant que les trois postulats.

Q : Je vois. Donc, si j'ai bien compris, tu es convaincu par les trois postulats, et les ASMs jouent le même rôle pour la thèse de Gurevich que les machines de Turing ont joué pour la thèse de Church, c'est bien ça ?

A : C'est un bon résumé.

Q : Maintenant que nous sommes enfin [*soupir*] d'accord sur la notion d'algorithme, nous pourrions passer au reste du titre. Tu précises « en temps quelconque, primitif récursif ou polynomial ». Je suppose que tu parles de la complexité en temps...

A : Exactement. Pour être un peu plus précis, comme les Abstract State Machines formalisent la notion intuitive d'algorithme séquentiel, je m'intéresse à `ASM` l'ensemble de ces machines, mais aussi à $\texttt{ASM}_{\mathcal{PR}}$ et $\texttt{ASM}_{\mathcal{P}ol}$, qui sont respectivement la restriction de `ASM` à des complexités en temps bornées par des fonctions primitives récursives ou par des fonctions polynomiales. Je détaille ces notions à la section 1.3 p.29.

Q : C'était rapide ! Il ne reste qu'à examiner « caractérisation impérative ». Tu essaies de faire coïncider `ASM`, $\texttt{ASM}_{\mathcal{PR}}$ et $\texttt{ASM}_{\mathcal{P}ol}$ avec des langages impératifs ?

A : Bien deviné ! J'ai essayé d'utiliser des présentations assez simples des langages impératifs, comme le langage `Loop` de Meyer et Ritchie dans [MR67] ou le langage `While` de Jones dans [Jon99], de façon à ce que mes travaux puissent être facilement adaptés aux langages de programmation usuels comme `C`, `Java` ou `Python` par exemple.

Q : Ah, en fait tu poursuis les travaux de ton directeur de thèse sur `APRA` ?

A : En effet, `APRA` était en fait $\texttt{ASM}_{\mathcal{PR}}$ mais restreint aux booléens et aux entiers unaires, et où seules les variables peuvent être mises à jour, en particulier il n'y a pas de tableau. Dans [APV10] Pierre a prouvé que si on ajoutait une commande `exit` au langage `Loop` alors le langage obtenu `LoopE` pouvait simuler `APRA`. Dans [MV09] afin de traduire son résultat dans les fonctions récursives primitives il a utilisé à la place le langage `LoopC` où les boucles sont conditionnées. Je détaille cela plus en détail à la section 2.2 p.43. Tu m'as demandé tout à l'heure s'il existait des langages de programmation qui soient algorithmiquement complets, et bien par exemple `LoopC` l'est pour `APRA`.

Q : Tu as l'air d'avoir apprécié le travail de ton directeur de thèse [*sourire*], mais quelle est ta propre contribution à tout cela ?

A : J'ai étendu le résultat aux tableaux et à toutes les structures de données possibles, de façon à ne plus restreindre les algorithmes étudiés. En particulier j'ai

soumis à Fundamenta Informaticae [Mar15a] un article montrant que `While` permet de simuler tous les algorithmes séquentiels. Mais je me suis également intéressé à la caractérisation des classes en temps, en particulier je montre p.103 l'équivalence algorithmique entre :

1. `ASM` et `While`
2. $\texttt{ASM}_{\mathcal{PR}}$ et `LoopC`
3. $\texttt{ASM}_{\mathcal{P}ol}$ et $\texttt{LoopC}_{stat}$

Q : Que désigne $\texttt{LoopC}_{stat}$?

A : Le langage `LoopC` où les bornes des boucles sont déterminées uniquement par l'état initial. J'ai discuté de ce langage en avril à la conférence DICE 2015 [Mar15b]. Je pense que c'est un candidat assez naturel pour clore la quête d'un modèle pour les algorithmes en temps polynomial, et je compte bientôt travailler avec Pierre pour l'adapter à un cadre fonctionnel.

Q : Voilà de bien belles promesses, il ne reste qu'à voir si tes preuves sont à la hauteur de tes ambitions...

Table des matières

Remerciements **3**

Discussion préliminaire **5**

1 Les algorithmes séquentiels **15**
1.1 Logique du premier ordre . . . 15
1.2 La thèse de Gurevich . . . 22
1.3 Taille et classes . . . 29

2 Modèles de calcul **35**
2.1 Les ASMs . . . 36
2.2 Les langages impératifs . . . 43
2.3 Simulation raisonnable . . . 52

3 Simulation des programmes impératifs **59**
3.1 Graphes d'exécution . . . 59
3.2 Traduction d'un programme impératif . . . 65
3.3 Temps primitif récursif . . . 74
3.4 Temps polynomial . . . 79

4 Simulation des ASMs **85**
4.1 Le programme noyau . . . 85
4.2 Le programme coquille . . . 92
4.3 Le programme de la simulation . . . 97

5 Conclusion **103**
5.1 Le théorème . . . 103
5.2 Critique . . . 105
5.3 Perspectives . . . 108

A Lemmes techniques **115**
A.1 Logique du premier ordre . . . 115
A.2 Les programmes impératifs . . . 121
A.3 Graphes d'exécution . . . 127
A.4 Traduction des ASMs . . . 130

B Compléments **139**
B.1 Les structures de données . . . 139
B.2 Une simulation plus équitable . . . 147

Bibliographie **159**

Index **163**

Résumé / Summary **165**

Chapitre 1

Les algorithmes séquentiels

Sommaire

1.1 Logique du premier ordre **15**
Structures du premier ordre 16
Termes d'un langage 18
Isomorphisme de structures 20
1.2 La thèse de Gurevich **22**
Premier postulat 22
Second postulat 24
Troisième postulat 26
1.3 Taille et classes **29**
Taille d'un élément et d'un état 30
Classes en temps 32

1.1 Logique du premier ordre

Bien que nous aurions pu l'éviter, dans un soucis de cohérence de notation et de simplification de la présentation, notre formalisation de la logique du premier ordre ne sera pas classique [1]. En effet :

1. Nous suivrons la présentation de Yuri Gurevich d'une part concernant le symbole *undef* pour les fonctions partielles, et d'autre part concernant les relations. Ainsi, les symboles de relation seront vus comme des symboles de fonction comme les autres, mais ayant leur valeur dans les booléens [2]. En particulier les formules seront vues comme des termes comme les autres.
2. Nous n'utiliserons pas la notion de variable logique dans les termes et formules. En particulier tous les termes et formules seront clos, et nous n'utiliserons pas les quantificateurs. Ce que nous appelerons « variables » seront les symboles dynamiques d'arité 0, dont l'interprétation pourra évoluer au fur et à mesure d'une exécution.

Ces précautions par rapport à la littérature habituelle étant prises, nous allons pouvoir présenter le plus succintement possible les structures du premier ordre, qui nous serviront dans la section suivante à formaliser (d'après le second postulat) les états d'une exécution :

1. ↑ Voir l'ouvrage de Cori-Lascar [CL03a] pour une présentation plus classique.
2. ↑ Dit autrement, un symbole de relation sera interprété par sa fonction caractéristique.

Structures du premier ordre

Un **langage** (du premier ordre) $\mathcal{L}$ est un ensemble de symboles de fonction.

Un langage sera dit **fini** si son cardinal est fini. Comme nous ne nous intéresserons qu'à des langages permettant d'écrire des programmes, nous supposerons que tous nos langages seront finis.

L'**arité** d'un symbole de fonction f est le nombre d'arguments de f. Par exemple la factorielle est unaire alors que l'addition est binaire.

Notation. Soient $\alpha(f)$ l'arité de f, ou tout simplement α en l'absence d'ambiguïté, et $F_\alpha(\mathcal{L})$ l'ensemble des symboles de fonction d'arité α de $\mathcal{L}$.

Pour homogénéiser les écritures, nous considèrerons que les symboles de constante c seront des symboles de fonction d'arité 0.

Définition 1.1.1. (Interprétation dans une structure)

Une structure (du premier ordre) X de langage $\mathcal{L}$, aussi appelée $\mathcal{L}$-structure, est la donnée :

1. d'un ensemble $\mathcal{U}(X)$ non vide appelé l'univers de X
2. pour chaque symbole $s \in \mathcal{L}$ d'une interprétation $\overline{s}^X$ telle que :
   a. si $c \in F_0(\mathcal{L})$ alors $\overline{c}^X$ est un élément de $\mathcal{U}(X)$
   b. si $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$
   alors $\overline{f}^X$ est une application : $\mathcal{U}(X)^\alpha \to \mathcal{U}(X)$

Notation. Soit $\mathcal{L}(X)$ le langage de la structure X.

Les fonctions d'un langage $\mathcal{L}$ sont interprétées dans une structure X comme étant totales. Pour simuler les fonctions partielles dans X nous suivrons la présentation de Gurevich [Gur00] en ajoutant un symbole *undef* :

Exemple 1.1.2. (Les fonctions partielles)

Nous supposerons que le symbole *undef* sera présent dans tous nos langages, et comme $\mathcal{U}(X)$ est non vide il contient au moins un élément, qui nous servira comme interprétation de *undef*.

Nous dirons que f n'est pas définie en $(a_1, \ldots, a_\alpha)$ si $\overline{f}^X(a_1, \ldots, a_\alpha) = \overline{undef}^X$, et nous noterons :

1. $Dom(f, X) =_{def} \{(a_1, \ldots, a_\alpha) \in \mathcal{U}(X)^\alpha \mid \overline{f}^X(a_1, \ldots, a_\alpha) \neq \overline{undef}^X\}$
2. $Im(f, X) =_{def} \overline{f}^X(Dom(f, X))$

Les fonctions partielles permettent notamment de définir des constructeurs et des opérations sur des sous-ensembles de l'univers. Je les utilise en particulier à la section B.1 pour introduire une notion de typage dans cette présentation. Cela n'est pas nécessaire pour prouver notre caractérisation impérative des classes algorithmiques, toutefois j'ai inclus cette section dans le chapitre des Compléments pour :

1. Obtenir une définition naturelle de la taille d'un élément dans une structure, en prenant le nombre de symboles utilisés pour représenter cet élément dans la structure. En effet, nous utilisons la notion de taille pour définir les classes en temps p.33, et les classes en espace p.74, et nous trouvions insatisfaisant de la supposer simplement donnée.

2. Esquisser avec la proposition p.143 que les types usuels en programmation sont implémentables dans ce formalisme, qui sinon pourrait sembler exotique. À l'inverse, les structures du premier peuvent sembler trop puissantes et nuire apparemment à notre théorème. Toutefois il est toujours possible de se restreindre aux structures acceptables. En particulier dans ce manuscrit nous n'utiliserons que les booléens et les entiers unaires.

Pour introduire les types nous devrons supposer que l'univers est assez grand, notamment nous aurons besoin de deux éléments supplémentaires pour $\overline{true}^X$ et $\overline{false}^X$, et d'un nombre dénombrable d'éléments pour les entiers unaires.

Exemple 1.1.3. (Les booléens $\mathbb{B}$)
L'univers contient deux nouveaux éléments distincts.
Les constructeurs sont $\{true, false\}$, interprétés par $\overline{true}^X \neq \overline{false}^X$.
Ajoutons les opérations $\neg$ et $\wedge$, définies uniquement sur les booléens :

1. $\overline{\neg}^X(\overline{true}^X) = \overline{false}^X$
   $\overline{\neg}^X(\overline{false}^X) = \overline{true}^X$
2. $\overline{\wedge}^X(\overline{true}^X, \overline{true}^X) = \overline{true}^X$
   $\overline{\wedge}^X(\overline{true}^X, \overline{false}^X) = \overline{false}^X$
   $\overline{\wedge}^X(\overline{false}^X, \overline{true}^X) = \overline{false}^X$
   $\overline{\wedge}^X(\overline{false}^X, \overline{false}^X) = \overline{false}^X$

$\{\neg, \wedge\}$ est un système complet de connecteurs pour le calcul propositionnel, donc permet d'obtenir les autres symboles usuels :

1. $(A \vee B) =_{def} \neg(\neg A \wedge \neg B)$
2. $(A \Rightarrow B) =_{def} (\neg A \vee B)$
3. $(A \Leftrightarrow B) =_{def} ((A \Rightarrow B) \wedge (B \Rightarrow A))$

Toutefois, à cause de la preuve de la proposition 2.1.9 p.40, nous n'aurons besoin que de $true, false, \neg, \wedge$ ainsi que des symboles de relation pour construire les formules p.19, donc nous n'utiliserons pas $\vee, \Rightarrow, \Leftrightarrow$.

Remarque. Les opérations $\neg, \wedge$ sont utilisées par confort d'écriture et pour ne pas présumer des capacités des modèles de calcul étudiés, mais en général il est possible de les simuler. Ainsi, en pseudo-code :

1. `if` $(F_1 \wedge F_2)$ `then` P_1 `else` P_2
   est simulable par :
   `if` F_1 `then` (`if` F_2 `then` P_1 `else` P_2) `else` P_2
2. `if` $\neg F$ `then` P_1 `else` P_2
   est simulable par :
   `if` F `then` P_2 `else` P_1

Comme il est possible de les simuler dans les deux modèles de calcul que nous définirons au second chapitre, en fait nous n'avons besoin à strictement parler que des constantes *true* et *false* et des relations pour construire les formules, mais nous ne nous servirons pas de cette simulation par soucis de lisibilité.

Remarque. (Symboles de relation)

Usuellement les symboles d'un langage sont distingués entre les fonctions et les relations, un symbole de **relation** R d'arité α étant interprété par un sous-ensemble : $\overline{R}^X \subseteq \mathcal{U}(X)^\alpha$.

Nous n'avons pas fait la distinction entre fonctions et relations car nous pouvons remplacer tout symbole de relation R par sa fonction caractéristique χ_R définie par :

$$\overline{\chi_R}^X(\vec{a}) =_{def} \begin{cases} \overline{true}^X & \text{si } \vec{a} \in \overline{R}^X \\ \overline{false}^X & \text{sinon} \end{cases}$$

Un langage $\mathcal{L}$ sera dit **égalitaire** s'il contient le symbole de relation $=$, interprété dans toute $\mathcal{L}$-structure X par la diagonale de $\mathcal{U}(X)$, c'est-à-dire l'ensemble $\{(a, a) \mid a \in \mathcal{U}(X)\}$. Afin de prouver la proposition 2.1.9 p.40, nous supposerons par la suite que tous nos langages seront égalitaires.

Si une relation R est binaire, nous abandonnerons la notation polonaise Rt_1t_2 pour nous conformer à l'usage avec la notation t_1Rt_2, en ajoutant des parenthèses si nécessaire. De plus, pour alléger les expressions nous noterons $t_1 \neq t_2$ le terme $\neg(t_1 = t_2)$.

Termes d'un langage

Les termes permettent de représenter syntaxiquement les éléments de l'univers considéré. Cette notion sera centrale pour énoncer le troisième postulat de Gurevich p.27, mais pas seulement.

En particulier, en utilisant notre formalisation des booléens, les formules seront des termes comme les autres. Cela simplifiera nos notations par la suite, en particulier la traduction p.88.

Rappel. Nous n'utilisons pas la notion de variable logique, le mot « variable » étant réservé aux symboles dynamiques d'arité 0. En particulier les termes seront clos, et nous n'utiliserons pas les quantificateurs dans les formules.

Cela ne sera cependant pas un problème car la preuve de proposition 2.1.9 p.40 montre qu'avec le troisième postulat nous n'avons en fait besoin que de formules de la forme $(\bigwedge_{1 \leq i,j \leq n} E_{ij})$ où E_{ij} est $t_i = t_j$ ou $t_i \neq t_j$, et t_i, t_j sont des termes.

Définition 1.1.4. (Termes d'un langage)

$Term(\mathcal{L})$, l'ensemble des termes (clos) du langage $\mathcal{L}$, est défini par induction :

1. $F_0(\mathcal{L}) \subseteq Term(\mathcal{L})$
2. Si $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et $t_1, \ldots, t_\alpha \in Term(\mathcal{L})$
   alors $ft_1 \ldots t_\alpha \in Term(\mathcal{L})$

Soit T un ensemble de termes. Nous dirons que T est **stable par sous-termes** s'il vérifie que pour tout $ft_1 \ldots t_\alpha \in T$ alors $t_1, \ldots, t_\alpha \in T$.

Nous définissons plus rigoureusement cette notation p.115 en introduisant $Sub(T)$, la clôture par sous-termes de T.

Cela nous sera utile pour faire des preuves par induction sur les termes, notamment l'isomorphisme par remplacement p.21.

Nous pouvons maintenant étendre l'interprétation des symboles dans une structure à l'interprétation des termes :

Définition 1.1.5. (Interprétation d'un terme)

L'interprétation $\overline{t}^X$ d'un terme $t \in Term(\mathcal{L})$ dans une $\mathcal{L}$-structure X est définie par induction sur t :

1. Si $t = c \in F_0(\mathcal{L})$
   Alors $\overline{t}^X =_{def} \overline{c}^X$
2. Si $t = ft_1 \dots t_\alpha$
   où $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et $t_1, \dots, t_\alpha \in Term(\mathcal{L})$
   Alors $\overline{t}^X =_{def} \overline{f}^X(\overline{t_1}^X, \dots, \overline{t_\alpha}^X)$

Illustrons cette définition en introduisant la représentation unaire des entiers, les autres bases étant esquissées p.144.

Exemple 1.1.6. (Les entiers unaires $\mathbb{N}_1$)

L'univers est étendu avec une copie de $\mathbb{N}$.

Les constructeurs sont $\{\emptyset, S\}$, interprétés par :

1. $\overline{\emptyset}^X = 0$
2. $\overline{S}^X : x \mapsto x + 1$

Où le successeur S est supposé indéfini ailleurs que sur $\mathbb{N}_1$.

Notez que les termes formés uniquement de constructeurs sont de la forme $S^n\emptyset$, et ils sont interprétés par $\overline{S^n\emptyset}^X = n$.

Nous dirons que $S^n\emptyset$ est la **représentation** de n.

Par défaut nous utiliserons dans notre manuscrit la représentation unaire pour les entiers. Ainsi, quand il n'y aura pas d'ambiguïté, nous confondrons la valeur et la représentation. Par exemple dans un programme 4 désignera le terme $SSSS\emptyset$, et $t+1$ sera une notation pour le terme St.

Enfin, nous introduisons p.142 une définition de la taille d'un élément comme étant le nombre de symboles utilisés dans sa représentation. Dans ce formalisme la taille de $n \in \mathbb{N}_1$ est de $n + 1$.

Remarque. Tous les termes ne sont pas des représentations.

Par exemple $4+3$, qu'il faudrait écrire rigoureusement $+S^4\emptyset S^3\emptyset$ (notation polonaise), vaut 7 mais n'est pas la représentation $S^7\emptyset$ de 7. Ainsi, dans notre formalisme la taille de $\overline{4+3}^X$ est donc 8 (le nombre de symboles formant $S^7\emptyset$), et non 10 (le nombre de symboles formant $+S^4\emptyset S^3\emptyset$).

Maintenant que nous avons introduit les booléens et les entiers unaires, il est possible de construire des termes en mélangeant les constructeurs ou les opérations disponibles. Par exemple, que vaut le terme $Strue$? Comme le successeur S est défini uniquement sur $\mathbb{N}_1$ et que $\overline{true}^X$ n'est pas un entier, nécessairement nous avons que $\overline{Strue}^X = \overline{undef}^X$.

Nous dirons par la suite qu'un terme t est **bien typé** sur X si $\overline{t}^X \neq \overline{undef}^X$.

Définition 1.1.7. (Formules d'un langage)

Une formule F est un terme ayant une forme particulière :

$$F =_{def} true \mid false \mid Rt_1 \dots t_n \mid \neg F \mid (F_1 \wedge F_2)$$

Remarque. Nous n'avons pas supposé que si $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ alors $\overline{f}^X(a_1, \dots, a_\alpha) = \overline{undef}^X$ s'il existe au moins un $a_i = \overline{undef}^X$.

En effet, dans notre présentation les symboles de relation n'ont que deux valeurs possibles : $\overline{true}^X$ et $\overline{false}^X$. Par exemple le terme $true = undef$ est évalué à $\overline{false}^X$ et non à $\overline{undef}^X$.

Ainsi, nous avons par induction immédiate sur F que toute formule est évaluée sur X par $\overline{true}^X$ ou $\overline{false}^X$. En particulier, une formule est toujours bien typée. Si $\overline{F}^X = \overline{true}^X$ nous dirons que F est vraie dans X, et si $\overline{F}^X = \overline{false}^X$ nous dirons que F est fausse dans X.

L'intéret de distinguer ainsi les formules des autres termes est de pouvoir écrire des conditionnelles ayant du sens (par exemple, que faire du programme `if` $Strue$ `then` $x :=$ 42 ?). Nous aborderons à nouveau la question au moment de définir la sémantique opérationnelle des ASMs p.37.

Isomorphisme de structures

Une des problématiques ouvertes dans notre champ d'étude est le questionnement de Gurevich dans [BDG09] sur l'existence d'une notion objective d'identité entre algorithmes. Nous soutiendrons sa réponse p.23 que deux algorithmes sont identiques s'ils ont les mêmes exécutions, mais avant cela nous devons fixer quand est-ce que deux structures sont identiques.

L'idée est qu'une structure représente l'état de la mémoire, et que d'un point de vue algorithmique il n'y a pas lieu de se préoccuper de détails particuliers d'implémentation, que seul le comportement algorithmique importe.

Exemple 1.1.8. (α-conversion)

Si dans un algorithme nous changeons le nom d'une variable par un nom frais, il est communément admis que l'algorithme obtenu est le même.

Pourtant, si je change le nom d'une variable x en la totalité du contenu des livres du Seigneur des anneaux, en théorie rien n'est changé mais en pratique la plupart des compilateurs ne pourraient pas suivre.

Cela peut être vu comme de l'obscurcissement de code, et sans aller jusqu'à évoquer les techniques utilisées lors de concours nous pouvons citer l'exemple ludique de la version orang-outan du Brainfuck [1], qui est certes Turing-complète mais ne ressemble guère à un langage de programmation.

Cela étant dit, nous utiliserons bien la notion d'isomorphisme pour identifier des structures. En fait, nous considérons que toute notre présentation est « à isomorphisme près », c'est-à-dire que tous les résultats peuvent être obtenus quelle que soit l'implémentation concrète de nos structures.

Pour pouvoir affirmer une telle chose nous devons supposer que l'ensemble des états est stable par isomorphisme (voir le second postulat p.24), et nous montrerons que les notions que nous définissons dans ce manuscrit sont préservées par isomorphisme.

Définition 1.1.9. (Isomorphisme entre structures)

Soient X et Y deux structures de même langage $\mathcal{L}$, et ζ une application : $\mathcal{U}(X) \to \mathcal{U}(Y)$.

ζ est un isomorphisme entre X et Y si :

1. ζ est surjective
2. Pour tout $c \in F_0(\mathcal{L})$: $\zeta(\overline{c}^X) = \overline{c}^Y$
3. Pour tout $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et pour tout $a_1, \ldots, a_\alpha \in \mathcal{U}(X)$:
   $\zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) = \overline{f}^Y(\zeta(a_1), \ldots, \zeta(a_\alpha))$

1. ↑ L'écrivain britannique Terry Pratchett, auteur des Annales du Disque-monde qui ont inspiré cette version, est mort le 12 mars 2015 durant la rédaction de notre manuscrit. Et pourtant la Tortue se Meut.

Remarque. Si ζ est un isomorphisme entre X et Y alors $\zeta(\overline{t}^X) = \overline{t}^Y$.

En utilisant le fait que nos langages sont égalitaires, nous prouvons avec le lemme A.1.5 p.117 qu'un isomorphisme est forcément injectif.

Remarque. Comme ζ est une application injective, nous avons :
$a = b \Leftrightarrow \zeta(a) = \zeta(b)$, en particulier (lemme A.1.6 p.117) :
F est vraie (resp. fausse) sur $X \Leftrightarrow F$ est vraie (resp. fausse) sur Y

Le fait que ζ soit injectif nous permet également de prouver avec le lemme A.1.8 p.117 que la définition donnée ici de l'isomorphisme entre structures est équivalente à la définition donnée dans une présentation plus classique, comme celle du Cori-Lascar [CL03a].

Comme ζ est une application injective et que nous avons supposé qu'elle était aussi surjective, nous avons donc que tout isomorphisme est bijectif. En particulier nous pouvons introduire ζ^{-1}, qui est également un isomorphisme :

Lemme 1.1.10. *(Inverse d'un isomorphisme)*
Si ζ est un isomorphisme entre X et Y
alors ζ^{-1} est un isomorphisme entre Y et X.

Démonstration. La preuve est faite p.118. □

Nous donnerons au fur et à mesure du manuscrit différents résultats « à isomorphisme près », comme la restriction de structure p.54 ou le typage p.140. Pour l'instant nous commençons par un lemme nécessaire à la thèse de Guervich, énonçant qu'en remplaçant dans une structure des éléments donnés par des éléments frais nous obtenons une structure isomorphe :

Définition 1.1.11. (Remplacement dans une structure)
Soit une structure X et deux ensembles $A \subseteq \mathcal{U}(X)$ et B tels qu'il existe une bijection φ entre A et B.

La structure Y obtenue en remplaçant dans X les éléments de A par les éléments de B est la structure de même langage $\mathcal{L}$ donnée par :

1. L'ensemble de base de Y est $(\mathcal{U}(X) \setminus A) \cup B$
2. Si $c \in F_0(\mathcal{L})$ alors :
$$\overline{c}^Y =_{def} \begin{cases} \varphi(\overline{c}^X) & \text{si } \overline{c}^X \in A \\ \overline{c}^X & \text{sinon} \end{cases}$$

En notant pour tout $b \in \mathcal{U}(Y)$ $b' = \varphi^{-1}(b)$ si $b \in B$ et $b' = b$ sinon :

3. Si $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et $b_1, \ldots, b_\alpha \in \mathcal{U}(Y)$ alors :
$$\overline{f}^Y(b_1, \ldots, b_\alpha) =_{def} \begin{cases} \varphi(\overline{f}^X(b'_1, \ldots, b'_\alpha) & \text{si } \overline{f}^X(b'_1, \ldots, b'_\alpha) \in A \\ \overline{f}^X(b'_1, \ldots, b'_\alpha) & \text{sinon} \end{cases}$$

Lemme 1.1.12. *(Isomorphisme par remplacement)*
Si $B \cap \mathcal{U}(X) = \varnothing$ alors la structure Y obtenue en remplaçant dans X les éléments de A par les éléments de B est isomorphe à X.

De plus, s'il existe un ensemble de termes T stable par sous-termes et tel que $\forall t \in T$ $\overline{t}^X \notin A$ alors $\forall t \in T$ $\overline{t}^X = \overline{t}^Y$.

Démonstration. La preuve est faite p.118 en utilisant comme isomorphisme $\zeta : \mathcal{U}(X) \to \mathcal{U}(Y)$ défini par :

$$\zeta(a) = \begin{cases} \varphi(a) & \text{si } a \in A \\ a & \text{sinon} \end{cases}$$

en montrant que $\zeta(a)' = a$.

La dernière partie du lemme se prouve par induction sur $t \in T$. □

1.2 La thèse de Gurevich

Tout le contenu de cette section, ainsi que celui de la section 2.1, est une reformulation de la thèse présentée par Yuri Gurevich dans [Gur00].

Nous entendons ici le mot « thèse » dans le même sens que la thèse de Church, qui affirme que les fonctions produites par des systèmes de calcul coïncident avec la notion intuitive de fonction calculable. Dans la mesure où cette notion est intuitive et non formelle il n'est pas possible de prouver la thèse de Church. Mais cette thèse est étayée par le fait que les différents systèmes de calcul (les fonctions récursives, le lambda-calcul et les machines de Turing) créés pour formaliser cette notion capturent le même ensemble de fonctions.

Ainsi, la thèse de Gurevich est que la notion intuitive d'algorithme est capturée par les Abstract State Machines (ASMs). Elle n'est pas prouvable, mais elle est étayée par une formalisation axiomatique de la notion d'**algorithme séquentiel**, et d'après le théorème 2.1 p.42 prouvant que les notions d'algorithme séquentiel et d'ASM coïncident.

Dans cette section nous reformulons donc la présentation axiomatique faite par Gurevich de la notion intuitive d'algorithme séquentiel.

Par algorithme séquentiel, nous entendons les algorithmes n'interagissant pas avec leur environnement [1] durant l'exécution. Cela exclut les algorithmes parallèles ou distribués, mais également les algorithmes non-déterministe en interprétant un générateur de hasard comme une communication particulière avec l'environnement.

Nous entendons également dans cette notion d'algorithme séquentiel que le calcul soit effectué étape par étape, et que durant chaque étape seule une quantité bornée de lecture et d'écriture puisse être faite.

Nous reviendrons sur ces notions au fur et à mesure de leur formalisation, qui sera faite à l'aide de trois postulats. Dans les autres chapitres nous appellerons algorithme les objets vérifiant ces trois postulats, et nous noterons `Algo` l'ensemble des algorithmes.

Premier postulat

Le premier postulat formalise l'idée de temps séquentiel [2], c'est-à-dire qu'un algorithme effectue un calcul étape par étape :

Postulat 1. (Temps séquentiel)

Un algorithme (séquentiel) est la donnée de :

1. Un ensemble d'états $S(A)$
2. Un ensemble d'états initiaux $I(A) \subseteq S(A)$
3. Une fonction de transition $\tau_A : S(A) \to S(A)$

1. ↑ Gurevich a étudié les algorithmes interactifs dans [Gur05].
2. ↑ Gurevich a également étudié les algorithmes en temps réel dans [GGV01].

Une **exécution** de A est une suite $\vec{X} = X_0, X_1, X_2, \ldots$ telle que :

1. X_0 est un état initial
2. Pour tout $i \in \mathbb{N}$: $X_{i+1} = \tau_A(X_i)$

Dans [BDG09] Gurevich discute de la difficulté voire l'impossibilité de pouvoir objectivement identifier deux programmes comme étant deux implémentations du même algorithme. Toutefois, le premier postulat a comme conséquence importante que deux algorithmes A et B seront vus comme **identiques** si :

1. $S(A) = S(B)$
2. $I(A) = I(B)$
3. $\tau_A = \tau_B$

Remarque. Deux algorithmes A et B ont le même ensemble d'exécutions si les deux dernières conditions uniquement sont remplies.

Plus précisément, si deux algorithmes A et B ont les mêmes exécutions alors $S(A) \neq S(B)$ ne peut arriver qu'à cause des états inaccessibles, et la condition $S(A) = S(B)$ peut donc être vue comme superflue.

Cependant cela n'aura pas d'incidence sur la suite, donc nous conservons la définition de Gurevich pour simplifier la présentation.

Il aurait été possible de définir un ensemble d'états terminaux $T(A)$ et de restreindre τ_A à $S(A) \setminus T(A)$. Cela nous aurait d'ailleurs été utile pour éviter l'introduction de la variable b_∞ dans la traduction p.65. Toutefois, là encore, nous suivrons l'exemple de Gurevich en simplifiant la présentation.

Au lieu de définir pour chaque algorithme un ensemble d'états terminaux, nous dirons donc plus généralement qu'un état X_m d'une exécution est final si $\tau_A(X_m) = X_m$. En effet dans ce cas l'exécution est :

$X_0, X_1, X_2, \ldots X_{m-1}, X_m, X_m, \ldots$

Et du point de vue d'un observateur extérieur l'exécution semblera arrêtée.

Remarque. Cette notion ne distingue pas l'arrêt d'une exécution d'une exécution bouclant indéfiniment sur le même état. De plus, il n'est pas possible de définir un mouvement ne faisant rien, puisqu'il sera interprété immédiatement comme étant final.

Une exécution sera dite **terminale** si elle contient un état final.

Lemme 1.2.1. *Si une exécution est terminale alors l'état final est unique.*

Démonstration. Soit $\vec{X}$ l'exécution terminale.

Supposons qu'il existe deux états X_m et X_n tels que $\tau_A(X_m) = X_m$ et $\tau_A(X_n) = X_n$.

m et n sont des entiers donc $m \leq n$ ou $n \leq m$.

Par exemple prenons $m \leq n$.

Comme $X_m = \tau_A^m(X_0)$ et $X_n = \tau_A^n(X_0)$, $X_n = \tau_A^{n-m}(X_m)$.

Or $\tau_A(X_m) = X_m$, donc $X_n = X_m$. □

Ainsi, dans l'exécution $X_0, X_1, X_2, \ldots X_{m-1}, X_m, X_m, \ldots$, si m est le plus petit entier tel que $\tau_A^m(X_0) = \tau_A^{m+1}(X_0)$ alors m sera appelé la **durée** de l'exécution $\vec{X}$. Si X est un état initial de l'algorithme A, nous noterons :

$$time(A, X_0) =_{def} \begin{cases} min\{i \in \mathbb{N} \mid \tau_A^i(X_0) = \tau_A^{i+1}(X_0)\} & \text{si } \vec{X} \text{ est terminale} \\ \infty & \text{sinon} \end{cases}$$

Nous utiliserons cette définition pour formaliser la notion de classe en temps p.33, en particulier $\mathcal{PR}$-time et $\mathcal{P}ol$-time que nous allons chercher à caractériser.

À présent, présentons sous forme de pseudo-code notre exemple récurrent : un algorithme efficace pour le *min* qui d'après Colson [Col91] ne peut être obtenu dans le modèle des fonctions récursives primitives `PR` :

TABLE 1.1 – *Min*, un algorithme efficace pour le minimum

```
Inputs : int m, int n
Init : int r := 0
  repeat
    if (r = m ∨ r = n)
    then stop
    else r := r + 1
Output : r
```

1. Les états doivent pouvoir mémoriser les valeurs de m, n et de r, donc pour l'instant notons-les par le triplet (m, n, r).
2. Les états initiaux sont les états où $r = 0$, c'est-à-dire de la forme $(m, n, 0)$, et ils sont distingués les uns des autres par les valeurs de m et n.
3. La fonction de transition τ_{Min} de cet algorithme peut être vue comme l'application $(m, n, r) \mapsto (m, n, r')$ où :

$$r' = \begin{cases} r & \text{si } r = m \text{ ou } r = n \\ r + 1 & \text{sinon} \end{cases}$$

Comme seul r a une valeur pouvant changer durant une exécution, celle-ci s'arrête quand $r = min(m, n)$, au bout de $min(m, n)$ étapes.

Second postulat

Comme le suggérait notre notation X, les états d'un algorithmes seront représentés par la suite comme des structures du premier ordre, sous certaines contraintes énoncées dans le second postulat.

Ce choix repose essentiellement sur l'analogie suivante : un symbole est à son interprétation ce que le nom d'un registre est à la valeur qu'il contient. En un sens, le concept d'interprétation peut être vu comme la version logique de concepts présent en informatique comme la notion de pointeur ou d'adressage.

Pour étayer ce point de vue, nous avons développé à la proposition B.1.10 143 un bestiaire des types usuels dans ce formalisme.

Dans la mesure où un algorithme est formalisé dans un langage, nous supposerons que tous les états seront des structures de ce langage. Comme un algorithme séquentiel est sans interaction avec son environnement durant le calcul, nous supposerons que la fonction de transition ne change pas l'univers des états. Enfin, les états seront toujours considérés indépendamment des détails d'implémentation, donc « à isomorphisme près » :

Postulat 2. (États abstraits)

1. Les états de A sont des structures du premier ordre
2. Tous les états de A ont le même langage $\mathcal{L}(A)$, supposé égalitaire et fini
3. τ_A conserve les univers des états
4. $S(A)$ et $I(A)$ sont clos par isomorphismes
5. Tout isomorphisme entre deux états X et Y est un isomorphisme entre $\tau_A(X)$ et $\tau_A(Y)$

En utilisant le second postulat et le lemme 1.2.1 p.23, nous montrons que la durée d'exécution est stable par isomorphisme :

Lemme 1.2.2. *(Isomorphisme du temps)*
Si $X, Y \in I(A)$ sont isomorphes alors $time(A, X) = time(A, Y)$.

Démonstration. Par hypothèse il existe un isomorphisme ζ tel que $\zeta(X) = Y$.
Or, d'après le second postulat, si $\zeta(X_i) = Y_i$ alors $\zeta(\tau_A(X_i)) = \tau_A(Y_i)$.
Donc par induction sur i : $\zeta(\tau_A^i(X)) = \tau_A^i(Y)$
Si $\vec{X}$ termine alors il existe X_m tel que $\tau_A(X_m) = X_m$.
Donc $\tau_A(Y_m) = \zeta(\tau_A(X_m)) = \zeta(X_m) = Y_m$, et l'exécution termine.
De même si $\vec{Y}$ termine, en utilisant ζ^{-1}.
Or, d'après le lemme 1.2.1 p.23, l'état final est unique.
Donc $time(A, X) = time(A, Y)$. □

Remarque. Comme l'ensemble d'états $I(A)$ est supposé stable par isomorphisme, il n'y a pas besoin de supposer que $Y \in I(A)$: l'isomorphisme entre $X \in I(A)$ et Y permet de le déduire.

Nous distinguerons dans $\mathcal{L}(A)$:

1. $Dyn(A)$: les **symboles dynamiques**, dont l'interprétation pourra être changée durant une exécution
2. $Stat(A)$: les **symboles statiques**, dont l'interprétation restera fixée durant une exécution

Pour un algorithme A, soient :

1. X un état de A
2. f un symbole de fonction dynamique d'arité α du langage de A
3. $a_1, \ldots, a_\alpha, b$ des éléments de l'univers de X

$(f, a_1, \ldots, a_\alpha, b)$ est une **mise à jour** de l'emplacement $(f, a_1, \ldots, a_\alpha)$.

Remarque. Nous avons présenté la logique du premier ordre à la section 1.1 en assimilant les symboles de constante à des symboles de fonction d'arité 0 et les symboles de relation à leur fonction caractéristique. Un des intérêts de cette démarche est que tous les symboles du langage sont des fonctions, ce qui nous permet à présent d'uniformiser la présentation des mises à jour.

Si u est une mise à jour, alors $X \oplus u$ [1] est une nouvelle structure de même langage et même univers tel que :

$$\overline{f}^{X \oplus u}(\vec{a}) =_{def} \begin{cases} b & \text{si } u = (f, \vec{a}, b) \\ \overline{f}^{X}(\vec{a}) & \text{sinon} \end{cases}$$

1. ↑ Nous notons $\oplus$ les mises à jour, et pas $+$ comme dans [Gur00] car l'associativité n'a pas de sens et surtout nous n'avons pas la commutativité : $(X \oplus (x, 0)) \oplus (x, 1) \neq (X \oplus (x, 1)) \oplus (x, 0)$.

Si $\overline{f}^X(\vec{a}) = b$ alors la mise à jour $(f, \vec{a}, b)$ sera dite triviale sur X car rien n'aura changé. En effet si $(f, \vec{a}, b)$ est triviale sur X alors $X \oplus (f, \vec{a}, b) = X$.

De même que pour $X \oplus u$ nous définissons une structure $X \oplus \Delta$ de même langage et univers que X, où Δ est un ensemble de mises à jour.

Δ sera dit **consistant** s'il ne contient pas deux mises à jour distinctes ayant les mêmes emplacements. En effet si $(f, \vec{a}, b), (f, \vec{a}, b') \in \Delta$ avec $b \neq b'$ alors Δ essaie d'attribuer deux valeurs distinctes à $f(\vec{a})$. Dans ce cas toutes les mises à jour échouent et rien n'est fait :

$$\overline{f}^{X \oplus \Delta}(\vec{a}) =_{def} \begin{cases} b & \text{si } (f, \vec{a}, b) \in \Delta \text{ et } \Delta \text{ est consistant} \\ \overline{f}^X(\vec{a}) & \text{sinon} \end{cases}$$

Lemme 1.2.3. *(Différence de structure)*

Si X et Y sont deux structures de mêmes langage et univers, alors il existe un unique ensemble consistant Δ de mises à jour non triviales tel que $Y = X \oplus \Delta$.

Démonstration. Comme X et Y ont les mêmes langage et univers, ils ont les mêmes emplacements. Il suffit de prendre :

$\Delta = \{(f, \vec{a}, b) \mid \overline{f}^Y(\vec{a}) = b \text{ et } \overline{f}^X(\vec{a}) \neq b\}$ □

Notation. Nous noterons $Y \ominus X$ ce Δ, ainsi : $X \oplus (Y \ominus X) = Y$

Pour un état $X \in S(A)$, nous noterons $\Delta(A, X) =_{def} \tau_A(X) \ominus X$ l'ensemble des mises à jour faites par A sur X.

Exemple 1.2.4. En reprenant notre algorithme *Min* p.24 :

1. Le langage est partagé entre :
   a. Un symbole dynamique r
   b. Les symboles statiques sont $m, n, =, \vee, S$ où seuls m et n dépendent de l'état initial. Par la suite nous dirons que $=, \vee, S$ sont des symboles avec une interprétation uniforme (voir p.29).
2. Nous avons que :
   a. $\overline{r}^{X_0} = 0$ si $X_0 \in I(A)$
   b. Pour tous les états X_i d'une exécution, les mises à jour sont toujours de la forme $(r, \overline{r}^X + 1)$.

   Ainsi, $\overline{r}^{X_i} = i$, jusqu'à l'état final où r est interprêté par :

$$time(Min, X_0) = min(\overline{m}^{X_0}, \overline{n}^{X_0})$$

Troisième postulat

Le premier postulat nous a permis de formaliser la notion de temps séquentiel, et le second nous a permis de représenter les états sous formes de structures. Cependant ils ne sont pas suffisants, puisqu'il nous manque le fait qu'à chaque étape un algorithme lit ou écrit une quantité bornée d'information.

Sans une telle restriction, il serait possible à un algorithme de faire une quantité de plus en plus importante d'actions élémentaires, ce qui peut être vu comme un comportement massivement parallèle, mais pas comme un comportement de calcul pas-à-pas.

Exemple 1.2.5. (Le lambda-calcul parallèle)
Le lambda-calcul est défini par :

$$\begin{array}{r} \text{Termes} : t :=_{def} x \mid \lambda x.t \mid (t_1)t_2 \\ \beta\text{−réduction} : (\lambda x.t_1)t_2 \to_\beta t_1[t_2/x] \end{array}$$

$t_1[t_2/x]$ est la substitution dans t_1 de toutes les occurrences libres de x par t_2. La β-réduction est ici la seule action élémentaire utilisée par le modèle, et un terme $(\lambda x.t_1)t_2$ est appelé un redex.

Prenons le terme $t = \lambda x.(x)x(x)x$. $(t)t$ est donc un redex, qui se β-réduit en $(t)t(t)t$. Le problème ici est qu'il y a maintenant deux redex, donc il faut un moyen de choisir quel redex réduire en premier.

Un tel choix est appelé une stratégie d'évaluation. Par exemple la stratégie par nom (voir par exemple la machine de Krivine [Kri07]) consiste à évaluer toujours le redex le plus à gauche, alors que la stratégie par valeur consiste à évaluer toujours le redex le plus à droite. Ces deux stratégies sont bien pas-à-pas car à chaque étape seulement une β-réduction est faite.

Prenons maintenant la stratégie parallèle, consistant à évaluer tous les redex en même temps :

$$\begin{array}{l} \quad (t)t \\ \to_{par} (t)t(t)t \\ \to_{par} (t)t(t)t(t)t(t)t \\ \to_{par} (t)t(t)t(t)t(t)t(t)t(t)t(t)t(t) \\ \to_{par} \cdots \end{array}$$

À la i-ième étape exactement 2^{i-1} β-réductions sont faites : cette stratégie n'est pas pas-à-pas.

Par la suite nous dirons que deux états X et Y **coïncident** sur un ensemble de termes T si pour tout $t \in T$: $\bar{t}^X = \bar{t}^Y$

Rappel. $Sub(T)$ est la clôture par sous-termes de T (voir p.115).

Postulat 3. (Exploration bornée)
Il existe un ensemble fini T de termes du langage de A tel que si deux états X et Y coïncident sur $Sub(T)$ alors $\Delta(A, X) = \Delta(A, Y)$.

Par la suite, T sera appelé un **témoin d'exploration bornée** de A, et les interprétations dans un état X des éléments de $Sub(T)$ seront appelées les **éléments critiques** de X.

Remarque. D'après le lemme A.1.3 p.116 nous aurions pu remplacer $Sub(T)$ par T dans le troisième postulat en supposant que T est déjà stable par sous-termes, comme cela est fait dans [Gur00]. Nous ne l'avons pas fait afin de rendre plus naturelle la définition du témoin d'exploration d'une ASM p.37.

Dans tous les cas nous avons besoin que X et Y coïncident sur un ensemble de termes stable par sous-termes, afin de pouvoir utiliser le lemme 1.1.12 p.21 dans la preuve suivante :

Lemme 1.2.6. *(Élements critiques)*
Si $(f, a_1, \ldots, a_\alpha, b) \in \Delta(A, X)$
alors $a_1, \ldots, a_\alpha, b$ *sont des éléments critiques de* X.

Démonstration. La preuve se fait par l'absurde.

Supposons qu'il existe une mise à jour $(f, a_1, \ldots, a_\alpha, a_0) \in \Delta(A, X)$ telle qu'un a_i ne soit pas un élément critique de X.

Soit b un élément frais, et Y la structure obtenue en remplaçant a_i par b dans X (voir la définition 1.1.11 p.21).

D'après le lemme 1.1.12 p.21, X et Y sont isomorphes. Comme X est un état et que Y est isomorphe à X alors d'après le second postulat p.24 Y est également un état.

Comme a_i n'est pas critique, d'après le le lemme 1.1.12 X et Y ont les mêmes interprétations en dehors de a_i, d'où X et Y coïncident sur $Sub(T)$. Donc d'après le troisième postulat $\Delta(A, X) = \Delta(A, Y)$.

Or $a_i \notin \mathcal{U}(Y)$ donc $(f, a_1, \ldots, a_\alpha, a_0)$ n'apparaît pas dans $\Delta(A, Y)$.

Mais cela contredit que $(f, a_1, \ldots, a_\alpha, a_0) \in \Delta(A, X)$. □

Corollaire 1.2.7. $\Delta(A, X)$ *est borné.*

Démonstration. D'après le lemme 1.2.6 les éléments pouvant apparaître dans les mises à jour de $\Delta(A, X)$ sont des interprétations des symboles de $Sub(T)$.

Or d'après le troisième postulat T est fini donc d'après le lemme A.1.4 p.116 $Sub(T)$ est fini aussi.

Ainsi, il n'y a qu'un nombre fini d'éléments pouvant apparaître dans les mises à jour de $\Delta(A, X)$.

De plus, le langage $\mathcal{L}(A)$ est fini, donc il n'y a qu'un nombre fini de fonctions dynamiques.

Donc $\Delta(A, X)$ est borné. □

Exemple 1.2.8. Reprenons notre algorithme *Min* p.24 :

La fonction de transition ne dépend que des interprétations de m, n, r donc $T(Min) = \{m, n, r\}$ est un témoin d'exploration.

Attention, il ne faut pas en déduire que réunir les termes d'un programme suffit pour obtenir un témoin d'exploration. Gurevich donne cet exemple pour montrer que cela est insuffisant :

Exemple 1.2.9. (Précaution pour les témoins d'exploration)

Soit le programme P suivant en pseudo-code :

$$\texttt{if } Rn \texttt{ then } n := Sn$$

Il semblerait que le témoin d'exploration soit $T = \{n, Sn, Rn\}$.

Soit X l'état tel que :

1. $\overline{n}^X = 0$
2. $\overline{S}^X : a \mapsto a + 1$
3. $\overline{R}^X : a \mapsto \begin{cases} \overline{true}^X \text{ si } a \text{ est pair} \\ \overline{false}^X \text{sinon} \end{cases}$

Ainsi : $\Delta(P, X) = \{(n, 1)\}$.

Supposons qu'il existe un état Y de même univers tel que :

1. $\overline{true}^Y = \overline{false}^X$
2. $\overline{false}^Y = \overline{true}^X$
3. Les autres interprétations sont inchangées.

Attention ! Il ne s'agit pas ici d'un remplacement comme pour le lemme 1.1.12 p.21. Par exemple, nous avons que :

$$\overline{Rn}^Y = \overline{Rn}^X = \overline{true}^X = \overline{false}^Y$$

Ainsi, Rn est vraie pour X mais pas pour Y. Donc à cause de la remarque p.21 X et Y ne peuvent pas être isomorphes.

De fait, la sémantique de R a été inversée :

$$\overline{R}^Y : a \mapsto \begin{cases} \overline{true}^Y \text{ si } a \text{ n'est pas pair} \\ \overline{false}^Y \text{ sinon} \end{cases}$$

Ainsi : $\Delta(P, Y) = \{\}$.

Or, par hypothèse les deux états X et Y coïncident sur $T = \{n, Sn, Rn\}$, donc d'après le troisième postulat nous devrions avoir $\Delta(P, X) = \Delta(P, Y)$, ce qui n'est pas le cas.

En fait, T n'est pas un témoin d'exploration, contrairement à l'ensemble de termes $\{true, false, n, Sn, Rn\}$.

Cette exemple nous conduira à ajouter les symboles *true* et *false* en tant que termes lus dans la définition du témoin d'exploration d'une ASM p.37.

Dans cette définition nous inclurons également les termes écrits, car les termes lus ne suffisent pas :

Exemple 1.2.10. (Nécessité des termes écrits)

Soit le programme $P = x := y$.

Prenons des états X et Y tels que :

1. $\overline{x}^X = 0$, $\overline{x}^Y = 1$
2. $\overline{y}^X = 1 = \overline{y}^X$

X et Y coïncident sur l'ensemble des termes lus $\{y\}$, mais :
$\Delta(P, X) = \{(x, 1)\} \neq \{\} = \Delta(P, Y)$
Ainsi, le témoin d'exploration doit être $\{x, y\}$ et non $\{y\}$.

De façon plus générale, les termes écrits sont nécessaires dans le témoin d'exploration pour éviter d'inclure des mises à jour triviales dans l'ensemble des mises à jour faites par l'algorithme.

1.3 Taille et classes

Rappel. Le langage $\mathcal{L}(A)$ de l'algorithme A est partagé entre :

1. $Dyn(A)$: Les symboles dynamiques, pouvant être mis à jour.
2. $Stat(A)$: Les symboles statiques, fixés durant une exécution.

Toutefois, en reprenant notre algorithme *Min* p.26 nous constatons qu'il y a une une différence parmi les symboles statiques. En effet, les symboles m et n ont une interprétation dépendant de l'état initial, ce sont même les entrées de l'algorithme. Au contraire, des symboles comme le successeur S ou les opérations booléennes comme $\neg$ ou $\wedge$ ne dépendent pas vraiment de l'état initial, mais plutôt de l'univers choisi pour les interprétations.

Nous dirons qu'un ensemble de symboles $\mathcal{L} \subseteq \mathcal{L}(A)$ a une **interprétation uniforme** si pour tout état X et Y : $X|_{\mathcal{L}}$ est isomorphe à $Y|_{\mathcal{L}}$, où $X|_{\mathcal{L}}$ est la notation définie p.54 pour la restriction de la structure X à un sous-langage $\mathcal{L}$.

En cas de mise à jour non triviale, un symbole $f \in Dyn(A)$ ne vérifie plus les mêmes équations durant une exécution, donc les symboles uniformes sont nécessairement statiques.

Notons $Init(A)$ les **symboles initiaux** de A, c'est-à-dire l'ensemble des symboles statiques dont l'interprétation n'est pas uniforme.

Enfin, nous développerons à la section B.1 la distinction implicite que nous avons faite à la première section entre les constructeurs et les opérations. Ainsi, comme suggéré à la remarque p.19, parmi les symboles uniformes nous distinguerons les constructeurs des opérations afin de pouvoir former les représentations des éléments.

Ainsi, désormais le langage $\mathcal{L}(A)$ de l'algorithme A sera partagé entre :

1. $Dyn(A)$: Les symboles dynamiques.
2. $Init(A)$: Les symboles initiaux.
3. $Cons(A)$: Les constructeurs.
4. $Oper(A)$: Les opérations.

Toutefois la distinction entre $Cons(A)$ et $Oper(A)$ ne nous intéressera qu'à la section B.1 pour essayer de définir naturellement une taille pour les éléments. Dans le reste du manuscrit nous pouvons supposer que la taille est donnée et considérer uniquement que $X|_{Oper(A)\sqcup Cons(A)}$ et $Y|_{Oper(A)\sqcup Cons(A)}$ sont isomorphes, sans autre distinction.

En revanche, la distinction entre les symboles uniformes et les autres est importante, puisque seuls les symboles de $Dyn(A) \sqcup Init(A)$ dépendent effectivement de l'état initial : ce sont les **entrées de l'algorithme**.

Exemple 1.3.1. Notre algorithme *Min* p.24 possède trois entrées : deux statiques m et n, et une dynamique r.

En supposant que les symboles dynamiques ont une valeur par défaut ou doivent être initialisés dans l'algorithme même, il aurait été possible de ne garder que les symboles initiaux comme entrées de l'algorithme. Toutefois, dans un soucis de généralité, je n'ai pas souhaité faire cette hypothèse.

Taille d'un élément et d'un état

Dans le cas où les structures de données considérées sont **représentables** (voir la définition B.1.6 p.142) il est possible d'utiliser la représentation d'un élément pour définir sa taille, ce que nous faisons d'ailleurs pour les types usuels dans notre bestiaire p.143.

Nous avions déjà esquissé cette idée p.19 pour les entiers unaires en suggérant que $S^n\emptyset$ était la représentation de n. Ainsi, nous avons précisé à la remarque p.19 que la taille de $\overline{4+3}^X$ était donc 8 (le nombre de symboles formant $S^7\emptyset$), et non 10 (le nombre de symboles formant $+S^4\emptyset S^3\emptyset$).

Nous n'écartons pas a priori la possibilité qu'un algorithme utilise un type de donnée exotique ne rentrant pas dans notre formalisme [1]. Afin que les questions de classes en espace et en temps aient du sens nous supposerons qu'à défaut d'être représentables les états soient au moins **mesurables**, c'est-à-dire que la fonction de taille $|.|$ définie p.142 et à valeur dans $\mathbb{N}_1^\star$ puisse être étendue aux éléments non représentables, tout en restant stable par isomorphisme.

1. ↑ Bien que pour nos ordinateurs numériques il est difficile d'imaginer un type de donnée n'étant pas représentable. Pour les ordinateurs analogiques il est généralement possible de se référer à des grandeurs physiques pour donner des estimations de taille. Par exemple la température ou la tension pour les calculateurs analogiques habituels, la profondeur ou le nombre de cellules pour des réseaux, le nombre de qbits pour les calculateurs quantiques, etc.

Mesure ce qui est mesurable, et rend mesurable ce qui ne l'est pas. — (Galilée)

Les termes apparaissant dans un témoin d'exploration $T(A)$ de A (voir p.27) sont les seuls termes du programmes à pouvoir être lus ou écrits. Ainsi, il est tentant d'assumer que la mémoire d'un état est constituée uniquement des éléments critiques. Pour un état X de l'algorithme A, la taille de la mémoire devrait donc êtres déterminée par les $\{|\overline{t}^X| \mid t \in T(A)\}$.

Il n'en est rien. En fait les éléments de $T(A)$ déterminent bien le prochain ensemble de mises à jour, comme supposé avec le troisième postulat, mais pas le reste de l'exécution : des valeurs « explosives » peuvent apparaître.

Exemple 1.3.2. (Valeur « explosive »)

Soit l'ASM $x := x + f(x)$.

Un témoin d'exploration stable par sous-termes est : $\{x, f(x), x + f(x)\}$.

Soit X_0 l'état initial où $\overline{x}^{X_0} = 0$, et $\overline{f}^{X_0}$ vaut 1 partout sauf en 3 où elle vaut 100! : ce sera notre valeur « explosive ».

Itérons à présent l'ASM :

1. $\overline{x}^{X_0} = 0$, $\overline{f(0)}^{X_0} = 1$ et $\overline{f(0) + 0}^{X_0} = 1$
2. $\overline{x}^{X_1} = 1$, $\overline{f(1)}^{X_1} = 1$ et $\overline{f(1) + 1}^{X_1} = 2$
3. $\overline{x}^{X_2} = 2$, $\overline{f(2)}^{X_2} = 1$ et $\overline{f(2) + 2}^{X_2} = 3$
4. $\overline{x}^{X_3} = 3$, $\overline{f(3)}^{X_3} = 100!$ et $\overline{f(3) + 3}^{X_3} = 3 + 100!$

Soudain la taille de la mémoire vient d'exploser, d'une façon « imprévisible » pour l'état initial puisqu'il ne peut accéder à f qu'en 0.

Il a fallu attendre le résultat d'un calcul pour pouvoir accéder à notre valeur « explosive », donc il n'est pas raisonnable de pouvoir supposer à l'avance de l'endroit où elle sera.

Pourtant f n'a pas changé durant l'exécution donc la taille de f ne devrait pas avoir changé non plus.

Il faut donc ne pas se contenter des éléments critiques, et à la place considérer que la taille de f est de l'ordre du pire cas possible, par exemple le maximum ou la somme de ses valeurs.

La somme n'est malheureusement pas possible. Ici f est définie sur tout $\mathbb{N}$ et elle vaut 1 presque partout, donc la somme donnerait l'infini alors que f ne contribue au calcul que pour deux valeurs possibles : 1 et 100!.

Ainsi, il semble raisonnable de prendre $sup_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$ comme taille pour f dans l'état X. Dans notre exemple précédent nous avons $|f|_{X_0} = 100!$, ce qui est bien la « pire » contribution de f durant le calcul.

Notation. Soit $|f|_X =_{def} \sup\limits_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$.

Remarque. Si $c \in F_0(\mathcal{L}(A))$ alors $|c|_X = sup\{|\overline{c}^X|\} = |\overline{c}^X|$, donc cette définition convient bien quelque soit l'arité du symbole.

Nous discutons p.145 du fait qu'un symbole dynamique d'arité $\alpha > 0$ est une implémentation du concept de tableau dans ce formalisme. La taille du tableau correspond alors à la taille de la plus grande des cases possibles, similaire par exemple à la notion de tableau de type *int* pour dire qu'une case nécessite quatre octets.

Il n'est pas souhaitable de compter la taille de tous les symboles statiques pour déterminer la taille de l'état. En particulier, les constructeurs posent problème puisqu'en dehors des types ayant un nombre fini d'éléments (par exemple les booléens) ils sont sensés donner une infinité d'éléments dont la taille ne fera que grossir. Par exemple pour le successeur :

Exemple 1.3.3. $|S| = \underset{n\in\mathbb{N}_1}{sup}|\overline{S}(n)| = \underset{n\in\mathbb{N}_1}{sup}|n+1| = \underset{n\in\mathbb{N}_1}{sup}\{n+2\} = \infty$

Ainsi, nous ne compterons pas la taille des symboles uniformes (constructeurs et opérations) pour la taille d'un état, ne serait-ce que parce qu'elle est la même dans tous les états (voir le lemme B.1.8 p.142). Seules les tailles des symboles dynamiques et des symboles initiaux (ce que nous avons appelé les entrées de l'algorithme) compteront pour déterminer la taille de l'état.

Nous n'avons pas encore précisé comment déterminer la taille de l'état en fonction de la taille des entrées. Généralement la taille des entrées d'un état X est agrégée afin de fournir une valeur à $|X|$. La somme ou le maximum peuvent être des choix judicieux, quoiqu'équivalents. En effet, comme le langage est fini nous avons :

$$\underset{f\in Dyn(A)\cup Init(A)}{max}|f|_X \leq \sum_{f\in Dyn(A)\cup Init(A)}|f|_X$$

$$\sum_{f\in Dyn(A)\cup Init(A)}|f|_X \leq card(Dyn(A)\sqcup Init(A)) \times \underset{f\in Dyn(A)\cup Init(A)}{max}|f|_X$$

Toutefois il n'est pas nécessaire ici de faire d'hypothèse, bien que cela aurait pu un peu simplifier les notations des propositions p.77 et p.82 sur les classes en temps des programmes impératifs. Ainsi, $|X|$ ne sera donc qu'une notation par commodité pour le n-uplet suivant :

Définition 1.3.4. (Taille d'un état de A)

$$|X| =_{def} (|f|_X)_{f\in Dyn(A)\sqcup Init(A)}$$

$$\text{où } |f|_X =_{def} \underset{a_i\in\mathcal{U}(A)}{sup}|\overline{f}^X(\vec{a})|$$

Classes en temps

Soit $n = card(Dyn(A)\sqcup Init(A))$.

La complexité en temps c_A d'un algorithme A devrait être une application : $\mathbb{N}_1^n \to \mathbb{N}_1$ prenant $|X|$ en entrée et donnant $time(A, X)$ (défini p.22) en sortie.

Le problème est qu'il peut tout à fait exister deux états X et Y tels que $|X|$ et $|Y|$ soient identiques mais avec des durées d'exécution différentes :

Exemple 1.3.5. $|\overline{false}^X| = 1 = |\overline{true}^Y|$ mais l'exécution de $\{$`while` b $\{\}$;$\}$ peut être en une étape dans le cas $\overline{b}^X = \overline{false}^X$ mais infinie dans le cas $\overline{b}^Y = \overline{true}^Y$.

Ainsi, si nous voulons mesurer le temps d'exécution d'un algorithme A par une fonction c_A ne tenant compte que de la taille de la mémoire nous devrions en fait définir quelque chose de la forme :

$$c_A(|X|) = sup_{|X|=|Y|}time(A, Y)$$

En particulier la complexité en temps ne peut être qu'une borne [1] puisque $time(A, X) \leq c_A(|X|)$ mais sans être forcément égale, comme nous l'avons vu dans l'exemple précédent.

Définition 1.3.6. (Classe en temps)
L'algorithme A est en temps $\mathcal{C}$ s'il existe une fonction $c_A \in \mathcal{C}$ telle que pour tout $X \in I(A)$: $time(A, X) \leq c_A(|X|)$.
Soit $\texttt{Algo}_{\mathcal{C}}$ l'ensemble des algorithmes $\mathcal{C}$-time.

Remarque. D'après la thèse de Church, en posant que $c_A(|X|) = \infty$ est indéfinie alors $\texttt{Algo} = \texttt{Algo}_{\mathcal{R}ec}$ où $\mathcal{R}ec$ est l'ensemble des fonctions récursive [2].

Nous nous intéresserons dans ce manuscrit en particulier à deux classes :

1. $\mathcal{PR}$ l'ensemble des **fonctions primitives récursives**, défini par :
   a. il contient la fonction nulle, le successeur et les projections
   b. il est clos par composition et récursion

   Pour une présentation plus détaillée voir [CL03b].

Remarque. Si $|X|$ est un n-uplet et non la somme ou le maximum des tailles des entrées alors nous devrons utiliser la récursion mutuelle et non la récursion simple dans la preuve de la proposition 3.3.5 p.77.

2. $\mathcal{P}ol$ l'ensemble des **fonctions polynomiales**.
   En particulier, une fonction polynomiale de degré ≤ 1 sera dite linéaire.

Je ne détaillerai pas davantage ces fonctions, que je supposerais bien connues dans la suite du manuscrit.
Avoir une borne en temps et non une égalité [3] est important car cela permet d'avoir bien tous les algorithmes :

Exemple 1.3.7. Notre algorithme *Min* p.24 fait exactement :
$time(Min, X_0) = min(\overline{m}^{X_0}, \overline{n}^{X_0})$ étapes.
Notez que $min \in \mathcal{PR}$ mais $min \notin \mathcal{P}ol$. Pourtant, comme $min(m, n) \leq n$ nous dirons que cet algorithme est linéaire (notamment polynomial) en temps.

Le fait d'avoir une borne en temps est profondément lié à la capacité de sortir d'un programme une fois que le calcul est fait. C'est pour cela que dans le cadre impératif des boucles `loop` ne suffisent pas et qu'il faut des `loop` conditionnées ou un `exit` (comme dans [APV10]).

1. ↑ Contrairement à [APV10] car comme les entrées étaient uniquement dans $\mathbb{N}_1$ la fonction de complexité en temps pouvait prendre directement les entrées comme arguments, et non la taille des entrées.
2. ↑ Si les états sont représentables alors il existe un nombre fini d'éléments d'une taille donnée, donc il n'y a qu'un nombre fini d'exécutions à lancer pour calculer $c_A(|X|) = sup_{|X|=|Y|} time(A, Y)$, en sachant que si une exécution ne termine pas alors la complexité n'est pas définie pour ces tailles. Donc c_A devrait pouvoir être implémentée par une machine de Turing.
3. ↑ Cela n'importe pas pour les fonctions primitives récursives car si une fonction arithmétique est bornée par une fonction primitive récursive alors elle est elle-même primitive récursive. [Source ?]

Chapitre 2

Modèles de calcul

Sommaire

2.1 Les ASMs **36**
Définition des ASMs 36
Simulation d'un algorithme 38
Théorème de Gurevich 41
2.2 Les langages impératifs **43**
Sémantique opérationnelle 46
Composition de programmes 48
Mises à jour d'un programme impératif 51
2.3 Simulation raisonnable **52**
Variables temporaires 53
Dilatation temporelle 54
Définition de la simulation 56

À la première section nous définirons les Abstract State Machines de Gurevich, et nous prouverons son théorème `Algo` = `ASM` pour obtenir une occurrence des algorithmes séquentiels (définis à la section 1.2) sous forme d'un modèle de calcul. Ainsi, un modèle de calcul M sera dit algorithmiquement complet si M peut simuler `ASM`.

Par exemple dans [FZG10], Marie Ferbus-Zanda et Serge Grigorieff ont montré que le lambda-calcul non typé [1] est algorithmiquement complet, en acceptant la nature oraculaire des algorithmes. En effet, pour chaque opération disponible un symbole de constante doit être ajouté au lambda-calcul afin que la simulation soit juste.

Toutefois à la définition 2.3.5 p.56 nous n'insistons pas tant sur la simulation que sur la simulation mutuelle, afin de caractériser les classes algorithmiques. Mais pourquoi caractériser et pas seulement simuler ?

Exemple 2.0.8. D'après le théorème d'ultime obstination de Colson [Col91], notre algorithme *Min* p.24 ne peut être implémenté dans le modèle de calcul de la récursion primitive, même si le minimum est une fonction primitive récursive.

Pourtant notre algorithme peut être implémenté dans le modèle des fonctions récursives. En effet, soit F la formule $(r = m \vee r = n)$. Il est possible de définir la fonction caractéristique de F :

$$\chi_F(m, n, r) =_{def} \begin{cases} 1 \text{ si } r = m \text{ ou } r = n \\ 0 \text{ sinon} \end{cases}$$

1. ↑ La preuve utilise le combinateur de point fixe de Curry : $\theta_F = (\lambda x.Fxx)\lambda x.Fxx$.

L'opérateur de minimisation μ est défini de la façon suivante :
Soit f une fonction récursive (pouvant être partielle).
$\mu[f]$ n'est définie sur $\vec{n}$ que s'il existe un x tel que $f(\vec{n}, x) = 0$, et que $f(\vec{n}, i)$ est définie pour tout $0 \leq i < x$. Dans ce cas :

$$\mu[f](\vec{n}) =_{def} min\{x \in \mathbb{N} \mid f(\vec{n}, x) = 0\}$$

En utilisant l'opérateur de minimisation sur la fonction caractéristique de F nous obtenons alors le résultat voulu :

$$r = \mu[1 - \chi_F](m, n)$$

D'un point de vue impératif (voir la section 2.2), *Min* ne peut être obtenu avec un `loop`, mais peut l'être avec un `while`.

Ainsi, certes cet algorithme peut être implémenté dans un modèle strictement plus puissant, mais en temps que fonction récursive et plus en temps que fonction primitive récursive. La question est donc de déterminer la « clôture algorithmique » des fonctions primitives récursives, à savoir un modèle qui soit algorithmiquement complet mais qui ne donnerait que les fonctions primitives récursives.

David Michel et Pierre Valarcher l'ont trouvé dans [MV09] en assouplissant la condition de récursion. Dans un modèle impératif cela revient à utiliser des exceptions pour sortir des boucles `loop`. Ainsi, ils ont obtenu une caractérisation de `APRA`, l'ensemble des algorithmes séquentiels à temps primitif récursif et n'utilisant que les booléens et entiers unaires comme structure de données.

Notre manuscrit a pour but d'étendre ce résultat de deux manières :

1. Obtenir une caractérisation quelle que soit la structure de données.
2. Caractériser `Algo` lui-même, mais aussi $\texttt{Algo}_{\mathcal{PR}}$ et $\texttt{Algo}_{\mathcal{P}ol}$.

Avant de travailler sur la simulation, introduisons donc les ASMs de Gurevich et notre modèle pour les langages impératifs.

2.1 Les ASMs

Définition des ASMs

Rappel. Afin de définir correctement les ASMs, nous utilisons le fait que :

1. Tous les langages considérés sont égalitaire (voir p.18).
2. Les booléens sont introduits p.17, et les formules p.19.
3. La notion de mise à jour $(f, \vec{a}, b)$ est introduite p.25.

Définition 2.1.1. (Programme d'une ASM)

$$\begin{aligned} \Pi =_{def} & f t_1 \ldots t_\alpha := t_0 \\ & \mid \texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif} \\ & \mid \texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar} \end{aligned}$$

où :

1. f est un symbole de fonction dynamique d'arité α

2. $t_1, \ldots, t_\alpha, t_0$ sont des termes du langage
3. F est une formule

$ft_1 \ldots t_\alpha := t_0$ sera appelée une **commande de mise à jour**.

Par la suite (notamment dans les preuves, pour simplifier les notations) il nous arrivera de noter u cette commande, et $\overline{u}^X$ son interprétation $(f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)$.

Ainsi, $\overline{u}^X$ est bien une mise à jour au sens défini p.25, et si u désigne une commande de mise à jour et non une mise à jour, nous continuerons à utiliser la notation $X \oplus u$, mais cette fois pour signifier $X \oplus \overline{u}^X$.

Notation. Pour $n = 0$ une commande `par` $\Pi_1 \| \ldots \| \Pi_n$ `endpar` est un programme « vide ». Si la partie `else` d'un programme `if` est `par endpar` alors nous écrirons simplement `if` F `then` Π `endif`.

Définition 2.1.2. (Termes lus et écrits par une ASM)

L'ensemble $Read(\Pi)$ des termes lus par Π et l'ensemble $Write(\Pi)$ des termes écrits par Π sont définis par induction sur Π :

$$\begin{aligned} Read(ft_1 \ldots t_\alpha := t_0) &=_{def} \{t_1, \ldots, t_\alpha, t_0\} \\ Read(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}) &=_{def} \{F\} \cup Read(\Pi_1) \cup Read(\Pi_2) \\ Read(\texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}) &=_{def} Read(\Pi_1) \cup \cdots \cup Read(\Pi_n) \end{aligned}$$

À cause de la remarque p.28 nous ajouterons toujours $true, false$ à $Read(\Pi)$.

$$\begin{aligned} Write(ft_1 \ldots t_\alpha := t_0) &=_{def} \{ft_1 \ldots t_\alpha\} \\ Write(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}) &=_{def} Write(\Pi_1) \cup Write(\Pi_2) \\ Write(\texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}) &=_{def} Write(\Pi_1) \cup \cdots \cup Write(\Pi_n) \end{aligned}$$

Soit $T(\Pi) =_{def} Read(\Pi) \cup Upd(\Pi)$.

Définition 2.1.3. (Sémantique opérationnelle des ASMs)

Un programme d'ASM Π induit une fonction de transition :

$$\tau_\Pi(X) =_{def} X \oplus \Delta(\Pi, X)$$

où l'ensemble de mises à jour $\Delta(\Pi, X)$ est défini par induction sur Π :

$$\begin{aligned} \Delta(ft_1 \ldots t_\alpha := t_0, X) &=_{def} \{(f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)\} \\ \Delta(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}, X) &=_{def} \Delta(\Pi_i, X) \\ &\qquad \text{où } i = 1 \text{ si } F \text{ est vraie sur } X \\ &\qquad \text{et } i = 2 \text{ si } F \text{ est fausse sur } X \\ \Delta(\texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}, X) &=_{def} \Delta(\Pi_1, X) \cup \cdots \cup \Delta(\Pi_n, X) \end{aligned}$$

La sémantique du `par` est un ensemble de mises à jour faites simultanément, contrairement aux langages impératifs de la section suivante où les mises à jour ne peuvent qu'être séquentielles.

Remarque. La sémantique opérationnelle du `if` est bien définie car en prenant la précaution p.18 que les relations soient toujours bien typées (voir p.19) nous avons que $\overline{F}^X = \overline{true}^X$ ou $\overline{F}^X = \overline{false}^X$.

Lemme 2.1.4. *(Coïncidence des termes lus)*

Si X et Y coïncident sur $Read(\Pi)$ alors $\Delta(\Pi, X) = \Delta(\Pi, Y)$.

Démonstration. Par induction immédiate sur Π, en se rappelant que nous disons que F est vraie quand $\overline{F}^X = \overline{true}^X$ (voir l'exemple p.28). □

Remarque. Malgré ce que la notation $\Delta(\Pi, X)$ pourrait laisser croire, il est possible que cet ensemble puisse être inconsistant ou contenir des mises à jour triviales sur X. Ainsi nous avons que :

$$\Delta(\tau_\Pi, X) = \tau_\Pi(X) \ominus X = (X \oplus \Delta(\Pi, X)) \ominus X \subseteq \Delta(\Pi, X)$$

mais il n'y a pas toujours égalité.

C'est à cause de cela, et de l'exemple 1.2.10 p.29, que nous avons inclus $Upd(\Pi)$ dans la définition de $T(\Pi)$.

Toutefois cela ne posera pas de problème par la suite, puisque les $\Delta(\Pi, X)$ des programmes d'ASM sous forme normale (voir p.40) seront des ensembles consistants de mises à jour non triviales.

Avant de montrer que les ASMs sont identiques aux algorithmes séquentiels, reprenons notre exemple récurrent :

Exemple 2.1.5. Une ASM pour l'algorithme *Min* p.24 :

Rappelons que les interprétations $\overline{m}^X$ et $\overline{n}^X$ caractérisent l'état initial X choisi, et que $\overline{r}^X = 0$.

$$\Pi_{Min} = \texttt{if } \neg(r = m \vee r = n) \texttt{ then } r := r + 1 \texttt{ endif}$$

Simulation d'un algorithme

Montrons à présent de théorème de Gurevich : `Algo` = `ASM`.

L'intérêt principal de ce théorème est de faire coïncider la présentation axiomatique des algorithmes séquentiels avec la présentation opérationnelle des ASMs. Dit autrement, nous pourrons utiliser les ASMs comme une occurrence des algorithmes séquentiels sous forme de modèle de calcul. Cela nous sera utile notamment à la définition de notre simulation p.56 pour introduire la notion de complétude algorithmique.

Soit $A \in$ `Algo` et T un témoin d'exploration bornée, supposé stable par sous-termes (voir la remarque p.27). Il nous faut donc montrer qu'il existe un programme d'ASM Π_A tel que pour tout $X \in S(A)$ nous ayons $\Delta(\Pi_A, X) = \Delta(A, X)$. Commençons par une version plus faible de ce résultat :

Lemme 2.1.6. *Pour tout état $X \in S(A)$ il existe un programme Π_X ne contenant que des termes de T tel que $\Delta(\Pi_X, X) = \Delta(A, X)$.*

Démonstration. D'après le lemme 1.2.6 p.27 les éléments apparaissant dans $\Delta(A, X)$ sont des éléments critiques de X.

Donc pour toute $(f, a_1, \ldots, a_\alpha, b) \in \Delta(A, X)$ il existe des termes $t_1, \ldots, t_\alpha, t_0$ de T tels que $\overline{t_1}^X = a_1, \ldots, \overline{t_\alpha}^X = a_\alpha, \overline{t_0}^X = b$.

Ainsi la mise à jour $u = f t_1 \ldots t_\alpha := t_0$ induit $\{(f, a_1, \ldots, a_\alpha, b)\}$ comme ensemble de mises à jour.

Or, d'après le corollaire 1.2.7 p.28 $\Delta(A, X)$ est fini.

Donc, en faisant de même pour toutes les mises à jour de $\Delta(A, X)$, nous obtenons des programmes $u_1, \ldots, u_k$, et il suffit de prendre le programme $\Pi_X = \texttt{par } u_1 \| \ldots \| u_k \texttt{ endpar}$ pour que par définition $\Delta(\Pi_X, X) = \Delta(A, X)$.

De plus, par construction les termes de Π_X sont tous dans T. □

Le problème est que Π_X dépend de X, or il nous faut un programme Π_A pour tous les états $X \in S(A)$, qui sont potentiellement infinis.

Commençons par prouver que Π_X fonctionne également pour tous les états « similaires » à X.

Si ζ est un isomorphisme entre X et Y et $(f, \vec{a}, b)$ une mise à jour telle que $\vec{a}, b \in \mathcal{U}(X)$ alors nous noterons $\zeta((f, \vec{a}, b))$ la mise à jour $(f, \overrightarrow{\zeta(a)}, \zeta(b))$, et nous étendrons la notation à un ensemble de mises à jour $\zeta(\Delta)$:

Lemme 2.1.7. *(Isomorphisme d'ensembles de mises à jour)*
Si ζ est un isomorphisme entre X et Y alors $\zeta(\Delta(A, X)) = \Delta(A, Y)$

Démonstration. Rappelons que selon le lemme 1.2.3 p.26 :
$\zeta(\tau_A(X) \ominus X)$
$= \zeta(\{(f, \vec{a}, b) \mid \overline{f}^{\tau_A(X)}(\vec{a}) = b \text{ et } \overline{f}^{X}(\vec{a}) \neq b\})$
$= \{(f, \overrightarrow{\zeta(a)}, \zeta(b)) \mid \overline{f}^{\tau_A(X)}(\vec{a}) = b \text{ et } \overline{f}^{X}(\vec{a}) \neq b\}$
$= \{(f, \overrightarrow{\zeta(a)}, \zeta(b)) \mid \overline{f}^{\zeta(\tau_A(X))}(\overrightarrow{\zeta(a)}) = \zeta(b) \text{ et } \overline{f}^{\zeta(X)}(\overrightarrow{\zeta(a)}) \neq \zeta(b)\}$
d'après la remarque p.21
$= \zeta(\tau_A(X)) \ominus \zeta(X)$, d'où :
$\zeta(\Delta(A, X))$
$= \zeta(\tau_A(X) \ominus X)$
$= \zeta(\tau_A(X)) \ominus \zeta(X)$
$= \tau_A(\zeta(X)) \ominus \zeta(X)$ d'après le second postulat p.24
$= \tau_A(Y) \ominus Y$
$= \Delta(A, Y)$ □

Pour un état X, soit E_X la relation binaire définie sur les éléments de T par : $E_X(t_1, t_2)$ est vraie si et seulement si $\overline{t_1}^X = \overline{t_2}^X$.

Si $E_X = E_Y$ alors les états X et Y seront dits T-**similaires**.

Lemme 2.1.8. *(Similarité d'états)*
Si deux états X et Y sont T-similaires alors $\Delta(\Pi_X, Y) = \Delta(A, Y)$.

Démonstration. Soit la structure Y' obtenue en remplaçant dans Y les éléments apparaissant à la fois dans $\mathcal{U}(X)$ et dans $\mathcal{U}(Y)$ par des éléments neufs.

D'après le lemme 1.1.12 Y et Y' sont isomorphes (et identiques si $\mathcal{U}(X) \cap \mathcal{U}(Y) = \varnothing$), et de plus $\mathcal{U}(X) \cap \mathcal{U}(Y') = \varnothing$.

Soit Z la structure obtenue en remplaçant dans Y' les éléments critiques (voir p.27) de Y' par les éléments critiques de X.

Ainsi, X et Z coïncident sur T.

Comme $\mathcal{U}(X) \cap \mathcal{U}(Y') = \varnothing$, nous avons d'après le lemme 1.1.12 que Y' et Z sont isomorphes.

D'où Y et Z sont isomorphes, et comme Y est un état Z en est un aussi.

Donc d'après le troisième postulat nous avons $\Delta(A, X) = \Delta(A, Z)$.

D'après le lemme 2.1.6 nous avons :

1. $\Delta(\Pi_X, X) = \Delta(A, X)$
2. Π_X ne contient que des termes de T

Comme Π_X ne contient que des termes de T nous avons $Read(\Pi) \subseteq T$.

Or, X et Z coïncident sur T donc d'après le lemme 2.1.4 nous avons que $\Delta(\Pi_X, Z) = \Delta(\Pi_X, X)$.

Donc $\Delta(\Pi_X, Z) = \Delta(\Pi_X, X) = \Delta(A, X) = \Delta(A, Z)$.
Soit ζ l'isomorphisme entre Y et Z.
Comme $\zeta(\bar{t}^Y) = \bar{t}^Z$ nous avons : $\zeta(\Delta(\Pi_X, Y)) = \Delta(\Pi_X, Z)$
De plus, d'après le lemme 2.1.7 : $\zeta(\Delta(A, Y)) = \Delta(A, Z)$
Nous avons donc : $\zeta(\Delta(\Pi_X, Y)) = \Delta(\Pi_X, Z) = \Delta(A, Z) = \zeta(\Delta(A, Y))$
En appliquant ζ^{-1} aux deux membres de l'égalité, nous avons le résultat recherché : $\Delta(\Pi_X, Y) = \Delta(A, Y)$ □

Nous avons ramené la « similarité » d'états à une notion syntaxique (la relation E_X). Nous allons donc pouvoir trouver le programme Π_A qui, comme annoncé, sera sous « forme normale » :

Proposition 2.1.9. *Pour tout algorithme A il existe un programme d'ASM Π_A tel que pour tout $X \in S(A)$: $\Delta(\Pi_A, X) = \Delta(A, X)$*

Démonstration. Rappelons que T est fini. Soit $T = \{t_1, t_2, \ldots, t_n\}$.
Comme T est fini il n'y a qu'un nombre fini[1] c de relations E_X.
Donc il existe un ensemble fini d'états $\{X_1, \ldots, X_c\}$ tel que tout état Y est T-similaire à un et un seul X_i, et dans ce cas d'après le lemme 2.1.8 : $\Delta(\Pi_{X_i}, Y) = \Delta(A, Y)$.
Soit un état X de A.
Rappelons que les langages considérés sont tous égalitaires (voir p.18).
Notons F_X la formule suivante sous forme normale conjonctive :

$$(\bigwedge_{1 \leq i,j \leq n} E_{ij}) \text{ où } E_{ij} \text{ est } \begin{cases} t_i = t_j \text{ si } E_X(t_i, t_j) \text{ est vraie} \\ \neg\, t_i = t_j \text{ sinon} \end{cases}$$

Ainsi, F_X est vraie dans un état Y si et seulement si X et Y sont T-similaires.
Les $F_{X_1}, \ldots, F_{X_c}$ sont donc des **gardes**, c'est-à-dire que pour tout état il existe une et une seule F_{X_i} vraie dans cet état.
Il suffit donc de prendre comme Π_A le programme d'ASM suivant :

```
       if F_{X_1} then Π_{X_1}
else   if F_{X_2} then Π_{X_2}
⋮
else   if F_{X_c} then Π_{X_c}
       endif ... endif
```

En effet pour tout état Y, il existe un unique X_i T-similaire à Y, d'où seule F_i est vraie.
Donc $\Delta(\Pi_A, Y) = \Delta(\Pi_{X_i}, Y) = \Delta(A, Y)$. □

Rappel. La fonction de transition τ_Π d'un programme d'ASM Π est définie pour toute structure X par $\tau_\Pi(X) = X \oplus \Delta(\Pi, X)$.

Remarque. Un programme Π_A comme obtenu dans cette proposition est dit sous **forme normale**.

1. Comme $\Delta(\Pi_A, X) = \Delta(A, X) = \tau_A(X) \ominus X$, nous avons que $\Delta(\Pi_A, X)$ est toujours un ensemble consistant de mises à jour non triviales.
   En particulier, comme $\tau_{\Pi_A}(X) = X \oplus \Delta(\Pi_A, X)$, en utilisant l'unicité du lemme 1.2.3 p.26 nous avons que $\Delta(\Pi_A, X) = \tau_{\Pi_A}(X) \ominus X$.

1. ↑ c est égal au nombre de relations d'équivalence sur un ensemble à n éléments, donc au n-ième nombre de Bell : $B(n) = \frac{1}{e}\sum_{k=0}^{\infty} \frac{k^n}{k!}$

2. Nous aurions pu utiliser la présentation alternative :

```
par
        if  F_{X_1} then Π_{X_1}
        if  F_{X_2} then Π_{X_2}
        ⋮
        if  F_{X_c} then Π_{X_c}
endpar
```

Mais nous avons préféré réserver les `par` aux ensembles de mises à jour, afin d'insister sur le fait que les mises à jour sont faites simultanément, et utiliser des `else if` pour insister sur le fait que les F_{X_i} sont mutuellement exclusives.

Exemple 2.1.10. En prenant $T = \{m, n, r\}$ nous pouvons donner une forme normale à l'ASM p.38 pour l'algorithme *Min* p.24.

Le cardinal de T est $n = 3$, donc le nombre de Bell associé est $B(3) = 5$, et en effet il y a 5 conditionnelles possibles dans la forme normale 2.1 p.41.

TABLE 2.1 – *Min* sous forme normale

```
     if (m = n ∧ n = r ∧ r = m) then
        par endpar
else if (m ≠ n ∧ n ≠ r ∧ r = m) then
        par endpar
else if (m ≠ n ∧ n = r ∧ r ≠ m) then
        par endpar
else if (m = n ∧ n ≠ r ∧ r ≠ m) then
        par r := r + 1 endpar
else if (m ≠ n ∧ n ≠ r ∧ r ≠ m) then
        par r := r + 1 endpar
     endif ... endif
```

Remarque. Rappelons qu'à cause de la remarque p.28 nous ajoutons toujours *true* et *false* au témoin d'exploration.

Cependant nous n'avons pas besoin d'ajouter *true* et *false* aux termes apparaissant dans la forme normale conjonctive :

$$\left(\bigwedge_{1\leq i,j\leq n} E_{ij}\right) \text{ où } E_{ij} \text{ est } \begin{cases} t_i = t_j \text{ si } E_X(t_i, t_j) \text{ est vraie} \\ \neg\, t_i = t_j \text{ sinon} \end{cases}$$

Si nous l'avions fait dans l'exemple précédent il aurait fallu $B(5) = 52$ au lieu de $B(3) = 5$ conditionnelles.

Théorème de Gurevich

Définition 2.1.11. (Définition des ASMs)

Une Abstract State Machine (ASM) M de langage $\mathcal{L}$ est la donnée :

1. d'un programme d'ASM Π sur $\mathcal{L}$

2. d'un ensemble $S(M)$ de $\mathcal{L}$-structures closes par isomorphismes et τ_Π
3. d'un sous-ensemble $I(M)$ clos par isomorphismes
4. d'une application τ_M, qui est la restriction de τ_Π à $S(M)$

Par la suite, si les ensembles d'états et d'états initiaux ne sont pas ambigus, nous aurons tendance à confondre les ASMs et les programmes d'ASM.

Théorème. *(Gurevich, 2000)*
`Algo` = `ASM`

Démonstration. La preuve se fait par inclusion mutuelle :

`Algo` $\subseteq$ `ASM`

Soit A un algorithme.

Soit M l'ASM donnée par :

1. Le programme Π_A obtenu à la proposition 2.1.9 p.40
2. L'ensemble $S(M) = S(A)$
3. Le sous-ensemble $I(M) = I(A) \subseteq S(A) = S(M)$
4. L'application τ_M, qui est τ_{Π_A} restreinte à $S(A)$

Pour tout état $X \in S(A)$:

$\tau_A(X)$
$= X \oplus (\tau_A(X) \ominus X)$
$= X \oplus \Delta(A, X)$
$= X \oplus \Delta(\Pi_A, X)$
$= \tau_{\Pi_A}(X)$
$= \tau_M(X)$

Donc au sens de la p.23 : $A = M$

`ASM` $\subseteq$ `Algo`

D'après la définition 2.1.11 une ASM M vérifie les deux premiers postulats.

Soit Π le programme de M, et $T(\Pi) = Read(\Pi) \cup Upd(\Pi)$ (p.37)

Montrons que $T(\Pi)$ est un témoin d'exploration bornée pour M :

Soient deux états X et Y coïncidant sur $T(\Pi)$.

Montrons que $\Delta(M, X) = \Delta(M, Y)$:

$\Delta(M, X) = \tau_M(X) \ominus X = \tau_\Pi(X) \ominus X$, et de même : $\Delta(M, Y) = \tau_\Pi(X) \ominus Y$

Comme X et Y coïncident sur $Read(\Pi)$ alors d'après le lemme 2.1.4 p.37 : $\Delta(\Pi, X) = \Delta(\Pi, Y)$.

Comme $\tau_\Pi(X) = X \oplus \Delta(\Pi, X)$, nous avons d'après le lemme 1.2.3 p.26 que $\tau_\Pi(X) \ominus X$ est $\Delta(\Pi, X)$ privé des mises à jour triviales sur X, et de même pour Y.

Comme de plus X et Y coïncident sur $Write(\Pi)$, nous avons qu'une mise à jour de $\Delta(\Pi, X) = \Delta(\Pi, Y)$ est triviale sur X si et seulement si elle est triviale sur Y.

Donc : $\tau_\Pi(X) \ominus X = \tau_\Pi(Y) \ominus Y$

D'où M est bien un algorithme séquentiel.

□

Remarque. En utilisant les deux parties de l'inclusion, nous avons que :

1. Toute ASM est un algorithme.

2. Tout algorithme est identique à une ASM sous forme normale.

Donc, pour toute ASM il existe une ASM identique mais dont le programme est sous forme normale. Ainsi, nous pourrons toujours supposer par la suite que le programme d'une ASM est déjà sous forme normale. Cela nous servira notamment p.87 pour définir la traduction des ASMs.

2.2 Les langages impératifs

Meyer et Ritchie ont prouvé dans [MR67] que le langage `Loop` calcule exactement les fonctions primitives récursives :

$$\begin{aligned}
\text{Commandes} : c =_{def}\ & n := 0 \\
& \mid n := n+1 \\
& \mid n_1 := n_2 \\
& \mid \texttt{loop}\ n\ \{s\} \\
\text{Séquences} : s =_{def}\ & \epsilon \\
& \mid c; s \\
\text{Programmes} : P =_{def} & \{s\}
\end{aligned}$$

Il s'agit d'un langage impératif minimal n'utilisant que les entiers unaires esquissés p.19, la séquentialisation « ; » de commandes, et des boucles $\texttt{loop}\ n\ \{s\}$ signifiant « répétez n fois la séquence s ». Notez que pour obtenir ce comportement il faut que n ne soit pas mis à jour dans s.

Par exemple les programmes $Add_1(m,n) = \{r := m; \texttt{loop}\ n\ \{r := r+1\};\}$ et $Add_2(m,n) = \{r := n; \texttt{loop}\ m\ \{r := r+1\};\}$ calculent tous les deux la somme de m et n, mais le premier fait $O(n)$ étapes alors que le second fait $O(m)$ étapes : la même fonction est calculée par deux algorithmes différents.

Remarque. Avec les commandes $n := 0$ et $n := n+1$ il n'est possible d'augmenter la valeur d'une variable qu'en l'incrémentant étape par étape, donc au cours d'une exécution $\vec{X}$:

$$\overline{n}^{X_i} \leq \overline{n}^{X_0} + i$$

Il est toutefois possible d'écraser une valeur par une valeur éventuellement plus grande avec la commande $n_1 := n_2$, donc si les variables sont $n_1, \ldots, n_k$ au cours d'une exécution $\vec{X}$:

$$\overline{n_j}^{X_i} \leq \max_{1 \leq j \leq k} \overline{n_j}^{X_0} + i$$

Remarque. Dans un soucis de clarté, quand nous présenterons les programmes seuls nous le ferons avec indentation (notation inspirée du langage `Python`). Nous donnons l'exemple $Add_1(m,n)$ à la table 2.2 p.43.

TABLE 2.2 – Indentation pour les programmes impératifs

```
r := m
loop n
   r := r + 1
```

Pour passer de la notation usuelle à la notation par indentation il suffit d'interpréter :

1. la séquentialisation ; par « sautez une ligne »

2. le début de bloc { par « sautez une ligne et incrémentez l'indentation »
3. la fin de bloc } par « décrémentez l'indentation »

Pour la réciproque, il faut seulement faire attention à ajouter le bon nombre de ;} à la fin du programme pour fermer les accolades déjà ouvertes.

Nous savons (par exemple par [CL03b]) que si des relations sont primitives récursives (par exemple l'égalité) alors les formules formées avec ces relations sont également primitives récursives. Nous pouvons donc rajouter des commandes `if` F $\{s_1\}$ `else` $\{s_2\}$ au langage `Loop` sans calculer de nouvelles fonctions, bien qu'enlever le coût de simulation donne de nouveaux algorithmes :

Exemple 2.2.1. Nous présentons à la table 2.3 p.44 deux essais de programme pour l'algorithme *Min* p.24.

TABLE 2.3 – Implémentation de *Min* dans `Loop` + `if`

$r := 0$	$r := 0$
`loop` m	`loop` n
`if` $r \neq n$	`if` $r \neq m$
$r := r + 1$	$r := r + 1$

Le problème est que le premier fait $O(m)$ étapes alors que le second fait $O(n)$ étapes, et non $O(min(m, n))$. Bien sûr avoir le symbole $\leq$ dans le langage permettrait de faire le calcul en $O(1)$ étapes mais cela reviendrait quasiment à se donner la fonction *min* pour simuler la fonction *min*...

Le problème est en fait plus profond. Colson a démontré dans [Col91] qu'un modèle de calcul comme les fonctions primitives récursives ne pouvait implémenter notre algorithme *Min* p.24. En effet, passé un nombre fini d'étapes une récursion finit toujours par « s'obstiner » sur le même argument. Dans notre exemple précédent, le premier programme fait $O(m)$ étapes même si dans le cas où $n \leq m$ plus aucune mise à jour n'est faite à cause de la conditionnelle `if` $r \neq n$: le programme s'obstine inutilement sur m. Le second programme, lui, s'obstine inutilement sur n.

Afin d'éviter cette obstination, il faut donc pouvoir sortir de la boucle. Pour cela il est possible :

1. D'utiliser une commande `exit` comme dans [APV10] : le programme s'arrête dès que `exit` est atteint durant l'exécution.
2. De conditionner les `loop` comme dans [MV09] : les boucles `loop` n `except` F agissent normalement, à ceci près que le programme sort de la boucle si F devient vraie.

En fait, les deux méthodes sont algorithmiquement équivalentes [1], la première correspondant par exemple à l'usage du `return` dans le langage `C` alors que la seconde correspond davantage aux exceptions. Nous avons choisi la seconde car elle respecte les contextes (voir p.48) et donc est plus naturelle pour la composition de programmes.

Exemple 2.2.2. Nous implémentons à la table 2.4 p.45 notre algorithme *Min* p.24 dans un cadre impératif.

Ces deux programmes nécessitent bien $O(min(m, n))$ étapes. D'ailleurs, leurs exécutions sont identiques, donc d'après notre notion de simulation p.56 ces deux programmes devraient être identifiés comme donnant le même algorithme.

1. ↑ La preuve est esquissée dans [MV09] et nous a inspiré pour nos preuves p.93.

TABLE 2.4 – Implémentation de *Min* dans `LoopC`

$r := 0$	$r := 0$
`loop` m `except` $r = n$	`loop` n `except` $r = m$
$\quad r := r + 1$	$\quad r := r + 1$

Le langage `Loop` avec des conditionnelles et des boucles `loop` conditionnées sera noté `LoopC`. Il nous servira pour écrire les algorithmes ayant une complexité en temps primitif récursif ou polynomiale, mais comme il est terminal (voir le corollaire p.50) il ne peut être suffisant pour tous les algorithmes (ayant une complexité potentiellement récursives).

Nous utilisons donc également le langage `While` inspiré de celui de Neil Jones dans [Jon99]. La réunion de `While` et de `LoopC` sera notée `Imp`, et sera notre modèle pour les programmes impératifs. C'est un modèle « minimal » dans le sens où il ne contient que les structures de contrôle nécessaires à notre résultat p.103, mais aussi dans le sens où tous les langages de programmation usuels (comme `C`, `Java` ou `Python` par exemple) ont au moins une version de ces commandes.

> C'est pour cela que nous parlerons en toute généralité des programmes impératifs pour les programmes de notre modèle. Notre résultat est donc non seulement de caractériser des classes d'algorithmes, mais aussi de le faire dans un formalisme plus usuel que les ASMs.

La principale différence entre notre modèle et ces langages de programmation est que nous ne travaillons pas avec des structures de données particulières. En effet, les algorithmes sont essentiellement oraculaires, c'est-à-dire qu'ils sont donnés pour un ensemble donné d'opérations considérées comme « élémentaires ». Par exemple nous avons l'habitude d'appeler « algorithme d'Euclide » le calcul suivant :

$$pgcd(a, b) = \begin{cases} a & \text{si } b = 0 \\ pgcd(b, a \ mod \ b) & \text{sinon} \end{cases}$$

Mais en fait la version d'Euclide ressemblait davantage à cela :

$$pgcd(a, b) = \begin{cases} pgcd(a - b, b) & \text{si } a > b \\ a & \text{si } a = b \\ pgcd(a, b - a) & \text{si } a < b \end{cases}$$

En mettant de côté la question métaphysique de savoir quel est le « vrai » algorithme d'Euclide, cette appellation peut légitimement être appliquée aux deux calculs. Cependant, ils donnent des exécutions très différentes selon que le modulo est considéré comme une opération élémentaire, ou s'il faut se contenter de la soustraction. Certes, on pourrait arguer qu'il est possible de simuler le modulo avec la soustraction, mais l'impact en terme de temps de calcul resterait.

Ainsi, nous utiliserons les mêmes commandes de mise à jour $ft_1 \dots t_k := t_0$ que pour les ASMs, en ne faisant pas de supposition sur le langage utilisé par l'algorithme.

> Notre résultat peut donc être vu comme une caractérisation des commandes nécessaires pour contrôler le flux d'une exécution selon la classe en temps voulue, à structure de données près.

Toutefois, cette question ne nous a pas laissés indifférents, et nous avons esquissé p.143 un bestiaire des structures de données usuelles afin de convaincre le lecteur de la « faisabilité » de notre approche.

Sémantique opérationnelle

Définition 2.2.3. (Syntaxe des programmes impératifs)

$$\begin{array}{rl} \text{Commandes :} & c =_{def} f t_1 \dots t_k := t_0 \\ & \quad \mid \texttt{if } F \ \{s_1\} \texttt{ else } \{s_2\} \\ & \quad \mid \texttt{while } F \ \{s\} \\ & \quad \mid \texttt{loop } n \texttt{ except } F \ \{s\} \\ \text{Séquences :} & s =_{def} \epsilon \\ & \quad \mid c; s \\ \text{Programmes :} & P =_{def} \{s\} \end{array}$$

où :
- f est un symbole dynamique d'arité k
- $t_1, \dots, t_k, t_0$ sont des termes
- F est une formule
- n est une variable unaire n'étant pas mise à jour dans $\{s\}$

Notation. Soit :

1. `Imp` les programmes impératifs
2. `While` les programmes impératifs sans commande `loop`
3. `LoopC` les programmes impératifs sans commande `while`

Ainsi, les programmes de `LoopC` $\cap$ `While` sont ceux n'utilisant comme commandes que les mises à jour et les conditionnelles.

Notation. Le symbole ϵ est utilisé pour noter la séquence vide, mais par simplicité nous noterons $\{\}$ et non $\{\epsilon\}$ le programme vide.

Similairement aux programmes d'`ASM`, nous écrirons seulement `if` F $\{s\}$ pour la commande `if` F $\{s\}$ `else` $\{\}$.

Dans le style de Meyer et Ritchie, nous noterons simplement `loop` n $\{s\}$ une commande `loop` n `except` *false* $\{s\}$.

Pour définir la sémantique opérationnelle d'une boucle `loop` nous utiliserons pour chaque commande un compteur i unique[1]. Ils appartiennent au langage du programme, et seront toujours initialisés à 0 pour garantir que $0 \leq \bar{i}^X \leq \overline{n}^X$.

Ainsi, le $i < n$ de la définition 2.2.4 pourrait être remplacé par un $\neq$. En sachant que le langage est déjà supposé égalitaire, il n'y a pas d'hypothèse supplémentaire à poser. Toutefois nous garderons la notation $i < n$ par soucis de lisibilité.

Dans un soucis de minimalité, contrairement au `for` habituel, nous ne supposerons pas que la valeur du compteur puisse être lue, bien que `for`$(j = 0; j < n; j{+}{+})$ $\{s[j]\}$ puisse être simulé par $j := 0;$ `loop` n $\{s[j]; j := j + 1\}$. Cela nous permettra de simplifier un peu les notations, notamment dans la preuve p.50.

De plus, bien qu'étant de fait des symboles dynamiques les compteurs i ne seront changés que par la sémantique opérationnelle et non par une mise à jour. Ainsi, les i seront utilisés mais syntaxiquement ils n'apparaitront pas dans les programmes.

Notation. La séquence de commandes $c; s$ peut être étendue aux séquences de séquences $s_1; s_2$ par $\epsilon; s_2 =_{def} s_2$ et $(c; s_1); s_2 =_{def} c; (s_1; s_2)$.

1. ↑ Par exemple un compteur i peut être indicé par son « numéro de ligne », voir la définition de la longueur p.127.

La sémantique opérationnelle des programmes impératifs sera formalisée par un système de transition d'états. Un état du système sera une paire $P \star X$ telle que P soit un programme impératif, et X une structure. Les transitions sont déterminées uniquement par la commande de tête[1] et par la structure en cours :

Définition 2.2.4. (Sémantique opérationnelle des programmes impératifs)

$$\{f t_1 \dots t_k := t_0; s\} \star X \succ \{s\} \star X \oplus (f, \overline{t_1}^X, \dots, \overline{t_k}^X, \overline{t_0}^X)$$

$$\begin{aligned}
\{\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}; s_3\} \star X &\succ \{s_1; s_3\} \star X \\
&\quad \text{si } F \text{ est vraie dans } X \\
\{\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}; s_3\} \star X &\succ \{s_2; s_3\} \star X \\
&\quad \text{si } F \text{ est fausse dans } X \\
\{\texttt{while}\ F\ \{s_1\}; s_2\} \star X &\succ \{s_1; \texttt{while}\ F\ \{s_1\}; s_2\} \star X \\
&\quad \text{si } F \text{ est vraie dans } X \\
\{\texttt{while}\ F\ \{s_1\}; s_2\} \star X &\succ \{s_2\} \star X \\
&\quad \text{si } F \text{ est fausse dans } X \\
\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s_2\} \star X &\succ \{s_1; \texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s_2\} \\
&\quad \star X \oplus (i, \overline{i}^X + 1) \\
&\quad \text{si } i < n \text{ et } F \text{ est fausse dans } X \\
\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s_2\} \star X &\succ \{s_2\} \star X \oplus (i, 0) \\
&\quad \text{si } i = n \text{ ou } F \text{ est vraie dans } X
\end{aligned}$$

Chaque successeur est unique : le système de transition est **déterministe**.

Notation. Une succession de i étapes de transition sera notée $\succ_i$, et définie par induction sur i :

1. $P_1 \star X_1 \succ_0 P_2 \star X_2$ si $P_1 = P_2$ et $X_1 = X_2$
2. $P_1 \star X_1 \succ_{i+1} P_2 \star X_2$ s'il existe $P_3 \star X_3$
   tel que $P_1 \star X_1 \succ P_3 \star X_3$ et $P_3 \star X_3 \succ_i P_2 \star X_2$

Remarque. $\succ_0$ est $=$ et $\succ_1$ est $\succ$.

Par induction sur i si $P_1 \star X_1 \succ_i P_2 \star X_2$ et $P_2 \star X_2 \succ_j P_3 \star X_3$ alors $P_1 \star X_1 \succ_{i+j} P_3 \star X_3$, donc en un sens cette relation est **transitive**.

Seuls les états de la forme $\{\} \star X$ n'ont pas de successeur : ce sont les états terminaux.

Notation. S'il existe i et X' tels que $P \star X \succ_i \{\} \star X'$ alors nous dirons que P **termine** sur X, ce qui sera noté par $P \downarrow X$.

Comme le système de transition est déterministe, i et X' sont uniques. Par la suite nous noterons $time(P, X)$ ce i et $P(X)$ ce X'. Ainsi :

$$\text{Si } P \downarrow X \text{ alors } P \star X \succ_{time(P,X)} \{\} \star P(X)$$

Remarque. $time(P, X) = 0$ si et seulement si $P = \{\}$.

Par la suite nous dirons qu'un programme est terminal s'il termine sur tous ses états. Nous prouverons notamment p.50 que les programmes de `LoopC` (ceux sans commande `while`) sont terminaux. En attendant, illustrons la sémantique opérationnelle par un exemple d'exécution :

1. ↑ Dit autrement, la sémantique opérationnelle des commandes correspond à une pile, voir par exemple [Jon99] pour une présentation renforçant cette idée.

Exemple 2.2.5. (Exécution de P_{min})

Prenons l'un des programmes impératifs donné p.44 pour l'algorithme *Min* p.24 : $P_{min} = \{r := 0; \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r + 1;\};\} \in \texttt{LoopC}$

m et n sont des symboles statiques qui sont les entrées de l'algorithme. Pour simplifier les notations nous les remplacerons par leurs valeurs initiales, supposées ici à 2 et à 3.

r est un symbole dynamique qui servira de sortie à l'algorithme. Comme il y a une commande `loop` un compteur i est nécessaire. Les autres interprétations étant uniformes nous ne noterons un état que par la valeur de r et de i :

$$\begin{array}{rr}
 & \{r := 0; \texttt{loop}\ 3\ \texttt{except}\ r = 2\ \{r := r + 1;\};\} \star [i = 0] \\
\succ & \{\texttt{loop}\ 3\ \texttt{except}\ r = 2\ \{r := r + 1;\};\} \star [i = 0, r = 0] \\
\succ & \{r := r + 1; \texttt{loop}\ 3\ \texttt{except}\ r = 2\ \{r := r + 1;\};\} \star [i = 1, r = 0] \\
\succ & \{\texttt{loop}\ 3\ \texttt{except}\ r = 2\ \{r := r + 1;\};\} \star [i = 1, r = 1] \\
\succ & \{r := r + 1; \texttt{loop}\ 3\ \texttt{except}\ r = 2\ \{r := r + 1;\};\} \star [i = 2, r = 1] \\
\succ & \{\texttt{loop}\ 3\ \texttt{except}\ r = 2\ \{r := r + 1;\};\} \star [i = 2, r = 2] \\
\succ & \{\} \star [i = 2, r = 2]
\end{array}$$

Nous avons que :

$$time(P_{min}, X) = 1 + 2 \times min(\overline{m}^X, \overline{n}^X) + 1 = O(min(\overline{m}^X, \overline{n}^X))$$

Ce qui, comme annoncé p.44, est bien la complexité recherchée.

Composition de programmes

Notation. Soit s_P la séquence telle que $P = \{s_P\}$. La **composition** P_1P_2 de deux programmes impératifs P_1 et P_2 est définie par $P_1P_2 =_{def} \{s_{P_1}; s_{P_2}\}$.

Parfois nous noterons c le programme $\{c;\}$. En particulier, cP sera une notation pour le programme $\{c; s_P\}$.

Les règles de transition 2.2.4 sont de la forme $cP \star X \succ P'P \star X'$ où le programme P peut être vu comme un contexte.

Remarque. S'il existe P_1 tel que $cP_1 \star X \succ P'P_1 \star X'$ alors comme le système de transition ne regarde que la commande de tête et l'état en cours nous avons pour tout P_2 que $cP_2 \star X \succ P'P_2 \star X'$.

La substitution de P_1 par P_2 est appelée une **commutation de contexte**. Les commutations de contexte sont très utiles, mais elles ne peuvent pas être étendues à $\succ_i$ dans tous les cas, par exemple :

Exemple 2.2.6. (Commutation de contexte incorrecte)

$$\begin{array}{l}
P = \{i := 0; i := i + 1;\} \\
P_1 = \{i := i + 1; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\} \\
P_2 = \{i := 0; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\}
\end{array}$$

Dans le cas suivant la commutation de contexte fonctionne :

$$\begin{array}{rll}
PP_1 \star X & = \{i := 0; i := i + 1; i := i + 1; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\} & \star X \\
 & \succ \{i := i + 1; i := i + 1; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\} & \star X \oplus (x, 0) \\
 & \succ \{i := i + 1; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\} & \star X \oplus (x, 1) \\
 & = P_1 & \star X \oplus (x, 1) \\
PP_2 \star X & = \{i := 0; i := i + 1; i := 0; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\} & \star X \\
 & \succ \{i := i + 1; i := 0; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\} & \star X \oplus (x, 0) \\
 & \succ \{i := 0; \texttt{while}\ \mathit{true}\ \{i := i + 1;\};\} & \star X \oplus (x, 1) \\
 & = P_2 & \star X \oplus (x, 1)
\end{array}$$

Mais dans ce cas la commutation de contexte ne fonctionne pas :

$$\begin{array}{lll}
PP_1 \star X & = \{i := 0; i := i+1; i := i+1; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \\
& \succ \{i := i+1; i := i+1; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,0) \\
& \succ \{i := i+1; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,1) \\
& \succ \{\texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,2) \\
& \succ \{i := i+1; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,2) \\
& = P_1 & \star X \oplus (x,2) \\
PP_2 \star X & = \{i := 0; i := i+1; i := 0; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \\
& \succ \{i := i+1; i := 0; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,0) \\
& \succ \{i := 0; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,1) \\
& \succ \{\texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,0) \\
& \succ \{i := i+1; i := 0; \texttt{while}\ \mathit{true}\ \{i := i+1;\};\} & \star X \oplus (x,0) \\
& = P_1 & \star X \oplus (x,0) \\
& \neq P_2 & \star X \oplus (x,2)
\end{array}$$

La commutation de contexte fonctionne dans le premier cas mais pas le second car l'exécution ne doit pas « entrer » dans le contexte à remplacer. Ainsi, i doit être borné par $time(P_1, X)$:

Lemme 2.2.7. *(Commutation transitive de contexte)*
Soit $P_1 \downarrow X$ et $i \leq time(P_1, X)$.
S'il existe P', X' et P_2 tels que $P_1P_2 \star X \succ_i P'P_2 \star X'$
alors pour tout P_3 : $P_1P_3 \star X \succ_i P'P_3 \star X'$

Démonstration. La preuve est faite p.121 par induction sur $time(P_1, X)$. □

En fait par la suite nous n'utiliserons des commutations de contexte que pour le cas $i = time(P_1, X)$:

Corollaire 2.2.8. *(États intermédiaires)*
Si $P_1 \downarrow X$ alors pour tout P_2 : $P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$

Démonstration. $P_1 \downarrow X$
Donc $P_1 \star X \succ_{time(P_1,X)} \{\} \star P_1(X)$
En appliquant le lemme 2.2.7 au contexte $\{\}$:
$P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$ □

Ce corollaire est utile notamment pour prouver que la composition de programmes fonctionne comme attendue :

Proposition 2.2.9. *(Composition de programmes)*
$P_1P_2 \downarrow X$ si et seulement si $P_1 \downarrow X$ et $P_2 \downarrow P_1(X)$, tel que :

1. $P_1P_2(X) = P_2(P_1(X))$
2. $time(P_1P_2, X) = time(P_1, X) + time(P_2, P_1(X))$

Démonstration. Les deux côtés de l'équivalence sont prouvés p.123 en utilisant :

1. le lemme A.2.2 : si $P_1P_2 \downarrow X$ alors $P_1 \downarrow X$
2. le corollaire 2.2.8 : si $P_1 \downarrow X$ alors $P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$
3. et la transitivité de la transition.

□

En utilisant cette proposition nous pouvons facilement prouver que comme annoncé les programmes de `LoopC` sont terminaux :

Corollaire 2.2.10. *(Terminaison des programmes de* `LoopC`*)*
Si $P \in$ `LoopC` alors P est terminal.

Démonstration. Par induction sur P :

$\boldsymbol{P = \{\}}$
Par définition $\{\}$ est terminal.

$\boldsymbol{P = P_1 P_2}$
Montrons que si la proposition est vraie pour P_1 et P_2 alors elle est vraie aussi pour $P = P_1 P_2$:
Pour tout X, comme P_1 et P_2 sont terminaux nous avons $P_1 \downarrow X$ et $P_2 \downarrow P_1(X)$.
Donc d'après la proposition 2.2.9 : $P_1 P_2 \downarrow X$
D'où $P = P_1 P_2$ est terminal également.

Il ne reste alors plus qu'à prouver la proposition pour les commandes seules en utilisant l'hypothèse d'induction :

$\boldsymbol{P = f t_1 \ldots t_k := t_0}$
Pour tout X, d'après la définition 2.2.4 :

$$\{f t_1 \ldots t_k := t_0;\} \star X \succ \{\} \star X \oplus (f, \overline{t_1}^X, \ldots, \overline{t_k}^X, \overline{t_0}^X)$$

Donc $P = f t_1 \ldots t_k := t_0$ est terminal.

$\boldsymbol{P =}$ `if` $\boldsymbol{F\ P_1}$ `else` $\boldsymbol{P_2}$
Pour tout X, d'après la définition 2.2.4 :

$$\{\texttt{if}\ F\ P_1\ \texttt{else}\ P_2;\} \star X \succ P_i \star X$$

où $i = 1$ si F est vraie dans X, et $i = 2$ sinon.
D'après l'hypothèse d'induction, P_1 et P_2 sont terminaux, donc :

$$P_i \star X \succ_{time(P_i, X)} \{\} \star P_i(X)$$

D'où par transitivité :

$$\{\texttt{if}\ F\ P_1\ \texttt{else}\ P_2;\} \star X \succ_{1+time(P_i, X)} \{\} \star P_i(X)$$

Donc $P =$ `if` $F\ P_1$ `else` P_2 est terminal également.

$\boldsymbol{P =}$ `loop` $\boldsymbol{n}$ `except` $\boldsymbol{F\ P_1}$
D'après l'hypothèse d'induction, P_1 est supposé terminal.
Rappelons que les compteurs ne sont pas lus par le programme, ce qui nous permet de simplifier les notations en utilisant :

1. $time(P_1, P_1^{a-1}(X))$ et non $time(P_1, P_1^{a-1}(X) \oplus (i, a))$
2. $P_1^a(X) \oplus (i, a)$ et non $P_1(P_1^{a-1}(X) \oplus (i, a))$

dans l'exécution suivante :

$$\begin{array}{lrr}
 & \texttt{loop}\ n\ \texttt{except}\ F\ P_1 \star & X \oplus (i,0) \\
\succ & P_1\ \texttt{loop}\ n\ \texttt{except}\ F\ P_1 \star & X \oplus (i,1) \\
\succ_{time(P_1,X)} & \texttt{loop}\ n\ \texttt{except}\ F\ P_1 \star & P_1(X) \oplus (i,1) \\
\succ & P_1\ \texttt{loop}\ n\ \texttt{except}\ F\ P_1 \star & P_1(X) \oplus (i,2) \\
\vdots & & \\
\succ_{time(P_1,P_1^{a-1}(X))} & \texttt{loop}\ n\ \texttt{except}\ F\ P_1 \star & P_1^a(X) \oplus (i,a) \\
\succ & \{\} \star & P_1^a(X) \oplus (i,0)
\end{array}$$

où a est le premier $i \leq \overline{n}^X$ tel que F est vraie dans $P_1^a(X)$, et sinon $a = \overline{n}^X$
Donc $P = \texttt{loop}\ n\ \texttt{except}\ F\ P_1$ est terminal également.

□

Mises à jour d'un programme impératif

Notation. Le système de transition 2.2.4 est déterministe, ce qui signifie que si $i \leq time(P,X)$ [1] alors il existe un unique P' et un unique X' tel que : $P \star X \succ_i P' \star X'$. Notons $\tau_X^i(P)$ ce P' et $\tau_P^i(X)$ ce X', ainsi :

$$P \star X \succ_i \tau_X^i(P) \star \tau_P^i(X)$$

En particulier, si $i = time(P,X)$ alors $\tau_X^i(P) = \{\}$ et $\tau_P^i(X) = P(X)$.

Si $i > time(P,X)$ nous pouvons étendre la notation en supposant que $\tau_X^i(P) = \{\}$ et $\tau_P^i(X) = P(X)$, de façon assez similaire à la sémantique utilisée pour les ASMs.

Remarque. τ_P^i n'est pas une fonction de transition au sens du premier postulat p.22 car $\tau_P^i(X) \neq \tau_P^1 \circ \cdots \circ \tau_P^1(X)$. En effet :

$$\text{Pour une exécution } P_0 \star X_0 \succ P_1 \star X_1 \succ \cdots \succ P_{i-1} \star X_{i-1} \succ P_i \star X_i$$

$$\text{Nous avons } X_i = \tau_{P_{i-1}}^1(X_{i-1}) = \cdots = (\tau_{P_{i-1}}^1 \circ \cdots \circ \tau_{P_1}^1 \circ \tau_{P_0}^1)(X_0)$$

$$\text{et non } (\tau_{P_0}^1 \circ \cdots \circ \tau_{P_0}^1)(X_0)$$

La succession des mises à jour faites par P sur X est :

$$\tau_P^1(X) \ominus \tau_P^0(X), \tau_P^2(X) \ominus \tau_P^1(X), \ldots$$

Dans le système de transition 2.2.4 une structure ne peut être mise à jour que par une commande $ft_1 \ldots t_k := t_0$ ou par une boucle `loop`, et une seule mise à jour est faite par étape.

Ainsi pour tout i : $\tau_P^{i+1}(X) \ominus \tau_P^i(X)$ est vide ou est un singleton.

Définition 2.2.11. (Ensemble de mises à jour d'un programme impératif)

$$\Delta(P,X) =_{def} \bigcup_{i \in \mathbb{N}} \tau_P^{i+1}(X) \ominus \tau_P^i(X)$$

Remarque. Si $P \downarrow X$ alors $\Delta(P,X)$ est fini (voir le lemme A.2.3 p.124).

1. ↑ Si P ne termine pas sur X nous pouvons poser que $time(P,X) = \infty$.

Par la suite nous dirons que P est sans **écrasement** sur X si $\Delta(P, X)$ est consistant.

En effet, pour les programmes impératifs il y a un écrasement si nous donnons une valeur à une variable et que par la suite nous lui donnons une autre valeur. Dans notre formalisme cela signifie qu'il existe dans $\Delta(P, X)$ deux mises à jour $(f, \vec{a}, b)$ et $(f, \vec{a}, b')$ avec $b \neq b'$, ce qui est bien la définition p.26 du fait que $\Delta(P, X)$ soit inconsistant.

Proposition 2.2.12. *(Mises à jour sans écrasement)*
Si $P \downarrow X$ sans écrasement alors $\Delta(P, X) = P(X) \ominus X$.

Démonstration. La preuve p.125 est faite par induction sur $time(P, X)$:
$\Delta(P, X)$
$= (\tau_P^1(X) \ominus X) \cup \Delta(\tau_X^1(P), \tau_P^1(X))$
$= (\tau_P^1(X) \ominus X) \cup (P(X) \ominus \tau_P^1(X))$ d'après l'hypothèse d'induction
Si $\tau_P^1(X) \ominus X = \varnothing$ alors $\Delta(P, X) = P(X) \ominus \tau_P^1(X) = P(X) \ominus X$.
Sinon $\tau_P^1(X) \ominus X = \{u\}$ et $\Delta(P, X) = \{u\} \cup (P(X) \ominus (X \oplus u))$
Comme $\Delta(P, X)$ est consistant, $u \in P(X) \ominus X$
Donc $\Delta(P, X) = P(X) \ominus X$ d'après le lemme A.2.4. □

2.3 Simulation raisonnable

La plupart des travaux réalisés en calculabilité visent essentiellement à déterminer si un modèle de calcul peut ou non calculer une classe de fonctions donnée, comment un modèle peut calculer les mêmes fonctions qu'un autre, ou comment restreindre un modèle pour caractériser une classe de fonctions donnée. Il s'agit d'obtenir une sortie voulue en fonction d'une entrée donnée.

Généralement les recherches sur ce qu'il se passe entre l'entrée et la sortie visent seulement à évaluer les ressources. Dans le cas des algorithmes séquentiels, il s'agit essentiellement du temps et de l'espace utilisés, afin d'identifier des classes (comme `PTime` ou `LogSpace`) ou de discuter du compromis entre les deux ressources.

Nous avons souhaité prendre une démarche plus générale, consistant non à se focaliser sur l'état initial et l'état final (éventuellement en utilisant des mesures plus ou moins externes comme le temps ou l'espace), mais à s'intéresser à tous les états d'une exécution.

La question de déterminer si deux algorithmes donnés sont identiques ou non est extrêmement vaste et a été discutée notamment dans [BDG09]. Dans ce manuscrit, nous avons soutenu la réponse de Gurevich p.23 : deux algorithmes séquentiels sont identiques s'ils ont les mêmes états et la même fonction de transition.

Toutefois nous sommes un peu plus spécifiques : seules les exécutions elles-mêmes nous intéressent. Ainsi, nous avons remarqué que pour que deux algorithmes séquentiels aient les mêmes exécutions il suffisait qu'ils aient les mêmes état initiaux et la même fonction de transition : par définition les états inatteignables n'apparaissent jamais au cours d'une exécution, donc ils ne nous intéressent pas.

> *Être, c'est être perçu ou percevoir.* — (George Berkeley)

Ainsi, de notre point de vue, une identité entre algorithmes est une identité entre les exécutions de ces algorithmes. Contrairement à une fonction qui est une boîte noire dont

seul le résultat est visible, nous pouvons observer étape par étape les différents états de la mémoire utilisés par l'algorithme [1] ainsi que constater quand l'algorithme s'arrête.

Toutefois, lorsque les modèles de calcul sont trop différents obtenir une identité peut se révéler impossible. Par exemple les ASMs peuvent faire simultanément un nombre borné de mises à jour : nous appelons « degré de parallélisme » cette borne à la p.89. En comparaison, comme dit à la p.51 les programmes impératifs ne peuvent faire au plus qu'une mise à jour par étape.

Pourtant, nous allons quand même prouver notre caractérisation impérative des classes algorithmiques, en utilisant non des exécutions identiques mais des exécutions similaires. Bien sûr, plus restreinte est cette notion de similarité plus fort sera le résultat.

Nous avons donc travaillé à réduire les hypothèses au maximum, quitte parfois à compliquer un peu la présentation, mais maintenant que les preuves sont faites il n'est pas difficile d'assouplir les conditions pour obtenir une présentation un peu plus intuitive.

Variables temporaires

Exemple 2.3.1. (Simulation d'un `loop` par un `while`)

Le programme suivant contenant un `loop` :

$$\{\texttt{loop}\ n\ \{s\};\}$$

est généralement simulé avec un `while` par :

$$\{i := 0; \texttt{while}\ i \neq n\ \{s;\ i := i + 1;\};\}$$

Mais ce programme utilise une variable supplémentaire, donc le langage n'est pas conservé. Pourtant, en un sens, ces programmes sont similaires.

Un autre exemple, sur lequel nous reviendrons à la p.86, consiste à échanger les valeurs de deux variables x et y. Dans un cadre séquentiel cela est généralement fait à l'aide d'une variable temporaire v :

$$v := x; x := y; y := v$$

Ainsi, le langage $\mathcal{L}_1$ du programme simulant pourra être une extension du langage $\mathcal{L}_2$ du programme simulé : $\mathcal{L}_1 \supseteq \mathcal{L}_2$

La question est alors de savoir jusqu'à quel point il peut être raisonnable d'étendre le langage considéré. Comme tous les langages considérés sont finis, nous avons que $\mathcal{L}_1 \setminus \mathcal{L}_2$ doit être fini, mais ce n'est pas suffisant.

Étendre le langage avec un symbole de fonction d'arité > 0 permettrait éventuellement de stocker une quantité d'information potentiellement non bornée dans un seul symbole. Or, l'utilisation d'une telle variable d'environnement peut être considérée comme déraisonnable.

Heureusement, nous n'aurons besoin que de supposer que $\mathcal{L}_1 \setminus \mathcal{L}_2$ est un ensemble borné de variables (c'est-à-dire de symboles de fonction dynamiques d'arité 0), où la borne ne dépend que du programme à simuler.

La question de l'initialisation de ces variables se pose alors [2]. En effet sans contrainte supplémentaire nous pourrions décider d'initialiser une des variables avec le résultat du calcul par exemple, ce qui serait aberrant.

1. ↑ Exemple concret : l'utilisation du GNU Debugger lors de l'analyse d'un programme.
2. ↑ Et je remercie Julien Cervelle du LACL de me l'avoir posée.

Toutefois, dans ce manuscrit les seules valeurs dépendant de l'état initial que nous utiliserons seront les tailles des entrées[1], les autres valeurs pouvant être uniformes (voir p.29).

Comme $\mathcal{L}_1 \supseteq \mathcal{L}_2$ nous aurons qu'un état simulé de P_2 sera un état simulant de P_1, mais restreint au langage de P_2. Cette notion peut être définie dans le cadre de la logique du premier ordre :

Définition 2.3.2. (Restriction d'une structure)
Soient :

1. $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages tels que $\mathcal{L}_1 \supseteq \mathcal{L}_2$
2. X une $\mathcal{L}_1$-structure

La restriction de X à $\mathcal{L}_2$, notée $X|_{\mathcal{L}_2}$, est une $\mathcal{L}_2$-structure définie par :

1. $\mathcal{U}(X|_{\mathcal{L}_2}) = \mathcal{U}(X)$
2. Pour tout $f \in \mathcal{L}_2$: $\overline{f}^{X|_{\mathcal{L}_2}} = \overline{f}^X$

Comme annoncé à la p.20 notre présentation est toujours « à isomorphisme près ». Ainsi, nous montrons que la notion de restriction de structure est stable par isomorphisme :

Lemme 2.3.3. *(Isomorphisme par restriction)*
Soient :

1. *$\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages tels que $\mathcal{L}_1 \supseteq \mathcal{L}_2$*
2. *X et Y deux $\mathcal{L}_1$-structures*

Si X et Y sont isomorphes alors $X|_{\mathcal{L}_2}$ et $Y|_{\mathcal{L}_2}$ le sont aussi.

Démonstration. La démonstration p.119 découle directement de la définition. □

Nous étendons également la notion de restriction de structures aux ensembles de mises à jour :

Notation. $\Delta|_{\mathcal{L}} =_{def} \{(f, \vec{a}, b) \in \Delta \mid f \in \mathcal{L}\}$

Et nous montrons p.120 que :

1. si $\Delta = \Delta_1 \cup \Delta_2$ alors $\Delta|_{\mathcal{L}} = \Delta_1|_{\mathcal{L}} \cup \Delta_2|_{\mathcal{L}}$
2. si Δ est consistant alors $(X \oplus \Delta)|_{\mathcal{L}} = X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}}$

Dilatation temporelle

Exemple 2.3.4. (Machines de Turing)
À chaque étape de calcul d'une machine de Turing plusieurs actions sont effectuées, dépendant de l'état en cours et du symbole lu dans la cellule en cours :

1. L'état de la machine est mis à jour
2. La machine peut écrire un nouveau symbole dans la cellule en cours
3. La tête de lecture peut se déplacer à gauche ou à droite

La notion d'action élémentaire est fondamentalement arbitraire. Par exemple ici nous pourrions considérer que

- ou bien que trois actions distinctes sont effectuées, donc que trois étapes de calcul sont faites

1. ↑ En fait des bornes sur les tailles suffisent, voir les programmes obtenus à la p.92.

– ou bien qu'une multi-action comprenant ces trois actions est effectuée en une seule étape.

Soit M_3 la machine de Turing ayant besoin de trois étapes pour faire ces actions, et soit M_1 la machine de Turing « classique » faisant ces trois actions élémentaires en une seule étape.

Une exécution $\vec{X}$ de M_3 peut « correspondre » à une exécution $\vec{Y}$ de M_1 si toutes les trois étapes de M_3 l'état est le même que celui de M_1 :

$$\begin{array}{ccc} M_3 & & M_1 \\ \hline X_0 & = & Y_0 \\ X_1 & & \\ X_2 & & \\ X_3 & = & Y_1 \\ X_4 & & \\ X_5 & & \\ X_6 & = & Y_2 \\ X_7 & & \\ \vdots & & \vdots \end{array}$$

Imaginons à présent que M_3 et M_1 soient implémentées sur des machines telles que la machine de M_3 calculerait exactement trois fois plus vite. Dans ce cas, du point de vue d'un observateur extérieur les implémentations de M_3 et M_1 seraient indiscernables.

Ainsi, l'unité de temps est arbitraire.

En dehors du temps de Planck il semble délicat de pouvoir affirmer l'atomicité du temps, ce qui ne nous a pas empêché de travailler sur des algorithmes avec un temps séquentiel. Affirmer que l'unité de temps choisie est intrinsèquement arbitraire peut être vu comme une forme de stabilité par homothétie des exécutions. Un parallèle peut également être fait avec la relativité restreinte : pour un observateur extérieur il n'y a aucune différence entre deux mouvements sans accélération, donc ne différant que par une dilatation temporelle uniforme.

Sans aller plus loin dans ces questions fascinantes, nous nous contenterons dans notre manuscrit d'identifier deux exécutions $\vec{X}$ et $\vec{Y}$ ayant une dilatation temporelle uniforme, c'est-à-dire telles que pour tout étape i : $X_{d\times i} = Y_i$. Cela signifie qu'il faut d étapes de $\vec{X}$ pour simuler une étape de $\vec{Y}$.

Dans ce cas la simulation sera dite **pas-à-pas**, et strictement pas-à-pas si $d = 1$. Malheureusement, contrairement à l'exemple ci-dessus où toutes les machines M_1 pouvaient être simulées par des machines M_3 avec la même dilatation temporelle $d = 3$, dans notre simulation la valeur de d pourra dépendre du programme à simuler [1].

Remarque. Cette contrainte de dilatation temporelle n'est pas suffisante pour garantir que le comportement de terminaison soit le même pour le programme simulé et le programme simulant. En effet, en reprenant nos machines de Turing :

1. ↑ La proposition p.89 indique cependant que cette valeur ne dépend que du nombre de termes critiques (d'où peuvent être déduits le nombre de termes lus et le nombre d'états syntaxiquement reconnus) et du degré de parallélisme de l'algorithme.

$$\begin{array}{ccc} M_3 & & M_1 \\ \hline X_0 & = & Y_0 \\ \vdots & & \vdots \\ X_{3\times m+0} & = & Y_m \\ X_{3\times m+1} & & \\ X_{3\times m+2} & & \\ X_{3\times m+0} & = & Y_m \\ \vdots & & \vdots \end{array}$$

Nous avons que Y_m est un état terminal (voir p.23) pour l'exécution $\vec{Y}$, mais si $d > 1$ l'exécution $\vec{X}$ peut alterner indéfiniment entre des états différents tout en respectant la contrainte de dilatation temporelle $X_{d\times i} = Y_i$.

Ainsi, une condition de terminaison $time(P_1, X) = d \times time(P_2, X) + e$ est nécessaire dans notre définition d'une simulation raisonnable, où e sera également une constante ne dépendant que du programme à simuler.

Définition de la simulation

En résumé, nos trois contraintes pour identifier les exécutions sont :

1. La simulation ne requiert qu'un nombre borné de variables fraîches
2. Les exécutions concordent à dilatation temporelle près
3. Elles s'arrêtent dans le même ordre de grandeur d'étapes

Plus précisément, nous suivrons la définition suivante pour notre simulation :

Définition 2.3.5. (Simulation raisonnable)
Soient M_1 et M_2 deux modèles de calcul.
M_1 simule M_2 si :
pour tout programme P_2 de M_2 il existe un programme P_1 de M_1 tel que :

1. $\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2)$, et $\mathcal{L}(P_1) \setminus \mathcal{L}(P_2)$ est un ensemble fini de variables

et qu'il existe $d \in \mathbb{N}^\star$ et $e \in \mathbb{N}$ dépendants seulement de P_2 tels que
pour toute exécution $\vec{Y}$ de P_2 il existe une exécution $\vec{X}$ de P_1 vérifiant :

2. pour tout $i \in \mathbb{N}$: $X_{d\times i}|_{\mathcal{L}(P_2)} = Y_i$
3. $time(P_1, X_0) = d \times time(P_2, Y_0) + e$

Si M_1 simule M_2 et M_2 simule M_1 alors ces deux modèles de calcul seront dits **algorithmiquement équivalents**, ce que nous noterons par $M_1 \simeq M_2$.

Remarque. Appliquer la seconde condition pour $i = 0$ nécessite que les états initiaux soient les mêmes, aux variables temporaires près.

D'après la troisième condition, si l'exécution simulée nécessite n étapes alors l'exécution simulante nécessitera $O(n)$ étapes, donc notre notion de simulation préserve la classe de complexité en temps définie p.33.

Lemme 2.3.6.
La simulation est une relation de pré-ordre sur les modèles de calcul.
De plus, l'équivalence algorithmique est bien une relation d'équivalence.

Démonstration. En effet, nous avons que cette relation est :

réflexive

Un modèle de calcul a les mêmes programmes et exécutions que lui-même.

transitive

Si M_1 simule M_2 qui simule M_3 alors M_1 simule M_3 :

1. $\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2) \supseteq \mathcal{L}(P_3)$
   $\mathcal{L}(P_1) \setminus \mathcal{L}(P_3) = (\mathcal{L}(P_1) \setminus \mathcal{L}(P_2)) \cup (\mathcal{L}(P_2) \setminus \mathcal{L}(P_3))$ est un ensemble fini de variables fraîches.

Si $t_2 = d_1 \times t_1 + e_1$ et $t_3 = d_2 \times t_2 + e_2$

Alors $t_3 = d_2 \times (d_1 \times t_1 + e_1) + e_2 = (d_2 \times d_1) \times t_1 + (d_2 \times e_1 + e_2)$, d'où :

2. La dilatation temporelle est $d_3 = d_1 \times d_2$.
3. Le temps d'arrêt est $e_3 = e_1 \times d_2 + e_2$.

La simulation est donc bien un pré-ordre. De plus, la symétrie de $\simeq$ est posée par définition, donc $\simeq$ est une relation d'équivalence.

antisymétrique ?

La simulation n'est pas symétrique.

Par exemple comme la fonction d'Ackermann $Ack \in \mathcal{R}ec \setminus \mathcal{PR}$ (voir par exemple [CL03b]) nous avons que $\texttt{ASM}_{\mathcal{PR}} \subsetneq \texttt{ASM}$.

Dans le même ordre d'idée, l'exemple p.53 suggère (mais de façon incomplète) comment simuler `LoopC` avec `While` mais comme les programmes de `LoopC` sont tous terminaux (voir p.50) la réciproque n'est pas possible.

Toutefois ce n'est pas parce qu'il y a bisimulation qu'il y a identité au sens de Gurevich (voir p.23). Par exemple nous montrerons dans ce manuscrit que $\texttt{While} \simeq \texttt{ASM}$, mais les ASMs peuvent faire plusieurs mises à jour en une étape, contrairement aux programmes impératifs.

Pour que cette relation soit un ordre il faudrait quotienter les modèles de calcul par la relation d'équivalence $\simeq$.

□

Nous allons donc pouvoir utiliser le théorème de Gurevich p.42 : $\texttt{Algo} = \texttt{ASM}$ pour utiliser le modèle de calcul `ASM` en temps que représentant des algorithmes séquentiels pour notre simulation.

Comme annoncé à la p.35 nous ne chercherons pas les modèles M pouvant simuler $\texttt{ASM}_{\mathcal{C}}$, mais tels que $M \simeq \texttt{ASM}_{\mathcal{C}}$: nous cherchons à caractériser les algorithmes d'une classe donnée $\mathcal{C}$.

Les deux chapitres suivants sont chacun dédiés à prouver un des côtés de la bisimulation, afin d'arriver à notre théorème p.103 :

1. `While` caractérise les algorithmes séquentiels.
2. `LoopC` caractérise les algorithmes primitifs récursifs, si l'espace est lui-même primitif récursif.
3. `LoopC` caractérise les algorithmes polynomiaux, si les bornes des boucles ne sont pas mises à jour.

Chapitre 3

Simulation des programmes impératifs

Sommaire

3.1 Graphes d'exécution **59**
Lignes d'un programme 59
Propriétés des graphes d'exécution 62
3.2 Traduction d'un programme impératif **65**
Définition de la traduction 65
La simulation 72
3.3 Temps primitif récursif **74**
Complexité en espace 74
Un fragment primitif récursif 77
3.4 Temps polynomial **79**
Bornes statiques 79
Un fragment polynomial 82

3.1 Graphes d'exécution

Le but de ce chapitre est de prouver que les ASMs peuvent simuler les programmes impératifs, au sens de la définition 2.3.5 p.56. Je rappelle que dans le système de transition p.47 définissant la sémantique opérationnelle de nos programmes impératifs les états sont de la forme $P \star X$. Les deux systèmes utilisent les mêmes commandes de mise à jour, donc la simulation revient en grande partie à simuler la partie « programme » de $P \star X$.

Notre but est donc de construire à partir de P un programme d'ASM Π_P (voir la traduction 3.2.1 p.65) contenant non seulement les mêmes mises à jour, mais aussi capable de déterminer où en est l'exécution de Π_P par rapport au programme impératif d'origine P.

Lignes d'un programme

L'idée intuitive pour traduire un programme impératif en une ASM est de traduire séparément chaque commande, en ajoutant une sorte de pointeur d'instruction [1] pour passer d'une commande à l'autre tout en respectant la sémantique opérationnelle.

1. ↑ Les programmes de cette forme sont appelés les « Control State ASMs » (voir [Bö05]).

Exemple 3.1.1. Le programme P_{min} p.48

$$\begin{array}{ll} 0: & r := 0 \\ 1: & \texttt{loop}\ n\ \texttt{except}\ r = m \\ 2: & \quad r := r + 1 \end{array}$$

peut être traduit à la main, voir la table 3.1 p.60.

TABLE 3.1 – Traduction de P_{min}

```
par
   if line = 0 then
      par r := 0 || line := 1 endpar
   endif
||
   if line = 1 then
      if (i ≠ n ∧ r ≠ m) then
         par i := i + 1 || line := 2 endpar
      else
         par i := 0 || line := 3 endpar
      endif
   endif
||
   if line = 2 then
      par r := r + 1 || line := 1 endpar
   endif
endpar
```

Remarque. Le numéro d'une ligne est la longueur du programme (voir p.127) avant la commande actuelle, donc les numéros de ligne sont tous entre 0 et $length(P)$, et $line = length(P)$ est la fin du programme. Comme le nombre de valeurs est borné, à la place d'un compteur d'instruction dont la valeur va changer, un nombre fini de booléens $b_0, b_1, \ldots b_{length(P)}$ peuvent être utilisés.

Cette approche a été suggérée dans [GV12], et est adaptée pour un langage de programmation basé sur des numéros de ligne (par exemple utilisant des commandes `goto`) mais pas pour un langage de programmation structuré comme `Imp`. En effet, le numéro de ligne peut faire la distinction entre deux commandes qui sont pourtant identiques du point de vue de la sémantique opérationnelle :

Exemple 3.1.2. Nous distinguons à la table 3.2 p.61 les commandes $\boxed{x := x + 1}$ et $\boxed{x := x + 1}$ par un marquage extérieur.

Du point de vue de la sémantique opérationnelle ces commandes sont identiques. Ainsi, les programmes P_2 et P_4 ne sont distingués que par le marquage alors que leur opérationnalité est la même : nous aurions pu remplacer P_2 par P_4 sans changer la suite de l'exécution.

Toutefois en affirmant que $\boxed{x := x + 1}$ et $\boxed{x := x + 1}$ sont des lignes différentes du programme, c'est bien une telle distinction que nous sommes en train de faire, qui n'a donc aucun fondement du point de vue de la sémantique opérationnelle.

De même que pour les boucles, cette distinction sans fondement concerne également les conditionnelles (voir la table 3.3 p.61).

Table 3.2 – Syntaxe avec marquage

P_1	$\succ P_2$	$\succ P_3$	$\succ P_4$
`while` *true* $\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$	$\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$	`while` *true* $\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$	$\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$ `while` *true* $\boxed{x := x+1}$

Table 3.3 – Marquage des conditionnelles

P_1	$\succ P_2$	$\succ P_3$
`if` *true* $y := 0$ $\boxed{x := x+1}$ `else` $y := 1$ $\boxed{x := x+1}$	$y := 0$ $\boxed{x := x+1}$	$\boxed{x := x+1}$

P_1	$\succ P_2$	$\succ P_3$
`if` *false* $y := 0$ $\boxed{x := x+1}$ `else` $y := 1$ $\boxed{x := x+1}$	$y := 1$ $\boxed{x := x+1}$	$\boxed{x := x+1}$

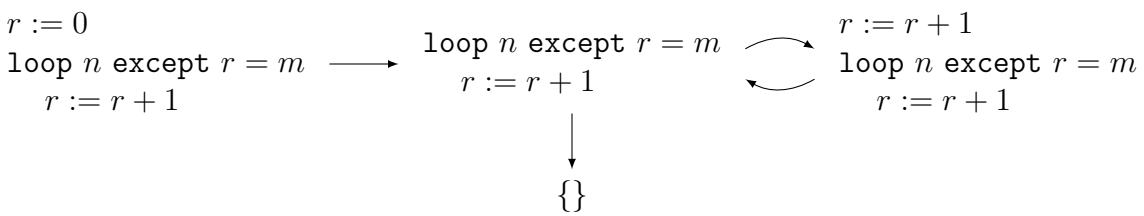
Ainsi, une traduction basée sur les numéros de ligne pourrait être définie, mais pour être distinguées les commandes devraient être marquées par leur profondeur (pour des boucles imbriquées) et le chemin de conditionnelles fait pour les atteindre. En plus d'être pénible cette approche serait inutile car ces commandes seraient de toute façon identiques pour la sémantique opérationnelle.

Nous allons donc non pas utiliser des booléens $\{b_i \mid 0 \leq i \leq length(P)\}$ indexés par les numéros de ligne, mais à la place nous allons les indexer par tous les programmes pouvant être rencontrés durant l'exécution :

Propriétés des graphes d'exécution

Les exécutions possibles d'un programme peuvent être représentées par un graphe où les arêtes sont les transitions possibles, et les sommets sont les programmes possibles :

Exemple 3.1.3. (Graphe d'exécution de P_{min})



Par la suite nous n'aurons besoin que de l'ensemble de sommets pour indexer les booléens, donc le graphe d'exécution de P_{min} sera noté par[1] :

$$\begin{aligned}\mathcal{G}(P_{min}) = \mathbf{\{}\ & \{r := 0; \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r+1;\};\},\\ & \{\texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r+1;\};\},\\ & \{r := r+1; \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r+1;\};\},\\ & \{\}\\ \mathbf{\}} & \end{aligned}$$

Notation. Pour définir formellement les graphes d'exécution nous introduisons la notation $\mathcal{G}(P_1)P_2$. Si $\mathcal{G}$ est un ensemble de programmes impératifs, et P un programme impératif, alors : $\mathcal{G}P =_{def} \{P_jP \mid P_j \in \mathcal{G}\}$

Remarque. D'après la définition de $\mathcal{G}P$:

1. $card(\mathcal{G}P) = card(\mathcal{G})$
2. $\mathcal{G}_1P \cup \mathcal{G}_2P = (\mathcal{G}_1 \cup \mathcal{G}_2)P$
3. $\mathcal{G}_1 \subseteq \mathcal{G}_2 \Rightarrow \mathcal{G}_1P \subseteq \mathcal{G}_2P$

Soit P un programme impératif. $\mathcal{G}(P)$ sera l'ensemble des $\tau_X^i(P)$ possibles, sans dépendre de l'état initial X choisi :

1. ↑ Dans un soucis de lisibilité, nous noterons dans cette section les accolades des ensembles en gras pour les distinguer des accolades des programmes.

Définition 3.1.4. (Graphe d'exécution) [1]

$$\begin{aligned}
\mathcal{G}(\{\}) &=_{def} \mathbf{\{\{\}\}} \\
\mathcal{G}(\{c; s\}) &=_{def} \mathcal{G}(c)\{s\} \cup \mathcal{G}(\{s\})
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}(t_1 \ldots t_k := t_0) &=_{def} \mathbf{\{\{}ft_1 \ldots t_k := t_0; \mathbf{\}\}} \\
\mathcal{G}(\texttt{if } F \texttt{ then } P_1 \texttt{ else } P_2) &=_{def} \mathbf{\{\{}\texttt{if } F \texttt{ then } P_1 \texttt{ else } P_2; \mathbf{\}\}} \cup \ \mathcal{G}(P_1) \cup \mathcal{G}(P_2) \\
\mathcal{G}(\texttt{while } F\ P_1) &=_{def} \mathcal{G}(P_1)\{\texttt{while } F\ P_1;\} \\
\mathcal{G}(\texttt{loop } n \texttt{ except } F\ P_1) &=_{def} \mathcal{G}(P_1)\{\texttt{loop } n \texttt{ except } F\ P_1;\}
\end{aligned}$$

Remarque. $\{\}, P \in \mathcal{G}(P)$

Montrons tout d'abord que les graphes d'exécution sont finis :

Lemme 3.1.5. *(Finitude d'un graphe d'exécution)*

$$card(\mathcal{G}(P)) \leq length(P) + 1$$

Démonstration. La preuve est faite p.127 par induction sur P, où la longueur est définie de manière usuelle. □

Ainsi, il n'y aura besoin que d'un nombre borné (et ne dépendant que de P) de booléens pour noter où le programme en est dans son exécution.

Remarque. Pour des programmes comme P_{min} dans l'exemple 3.1.3 p.62, la valeur

$$card(\mathcal{G}(P)) = length(P) + 1$$

peut effectivement être atteinte. En revanche les exemples p.60 montrent que l'inégalité ne peut être remplacée par une égalité. Donc cette borne est optimale.

Nous allons donc utiliser l'ensemble de booléens $\{b_{P_j} \mid P_j \in \mathcal{G}(P)\}$ pour contrôler à chaque étape où en est l'exécution. Pour que nous puissions utiliser cette ensemble dans notre traduction de P en Π_P, il faut prouver que si $P_j \in \mathcal{G}(P)$ alors le programme suivant $\tau^1_X(P) \in \mathcal{G}(P)$ également. Pour cela nous avons besoin de deux lemmes techniques :

Lemme 3.1.6. *(Composition de graphes d'exécution)*

$$\mathcal{G}(P_1 P_2) = \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$$

Démonstration. La preuve est faite p.128 par induction sur P_1. □

Lemme 3.1.7. *(Sous-graphes d'exécution)*
Si $Q \in \mathcal{G}(P)$ alors $\mathcal{G}(Q) \subseteq \mathcal{G}(P)$

Démonstration. La preuve est faite p.129 par induction sur P en utilisant le lemme 3.1.6. □

Nous pouvons maintenant prouver que si un programme apparaît dans $\mathcal{G}(P)$ alors ses successeurs possibles pour la sémantique opérationnelle y apparaissent également :

Proposition 3.1.8. *(Clôture opérationnelle des graphes d'exécution)*

1. ↑ `while` et `loop` ont la même définition, donc nous réduirons les preuves des graphes d'exécution en ne traitant que le cas du `while`, le cas `loop` étant identique.

1. *Si* $\{u; s\} \in \mathcal{G}(P)$
   Alors $\{s\} \in \mathcal{G}(P)$
2. *Si* $\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\} \in \mathcal{G}(P)$
   Alors $\{s_1; s\}$ *et* $\{s_2; s\} \in \mathcal{G}(P)$
3. *Si* $\{\texttt{while } F\ \{s_1\}; s\} \in \mathcal{G}(P)$
   Alors $\{s_1; \texttt{while } F\ \{s_1\}; s\}$ *et* $\{s\} \in \mathcal{G}(P)$
4. *Si* $\{\texttt{loop } n \texttt{ except } F\ \{s_1\}; s\} \in \mathcal{G}(P)$
   Alors $\{s_1; \texttt{loop } n \texttt{ except } F\ \{s_1\}; s\}$ *et* $\{s\} \in \mathcal{G}(P)$

Démonstration. La preuve est faite par cas :

$\boldsymbol{\{u; s\} \in \mathcal{G}(P)}$

Dans ce cas :
$\boldsymbol{\{}\{u; s\}\boldsymbol{\}} \cup \mathcal{G}(\{s\})$
$= \mathcal{G}(\{u; s\})$
$\subseteq \mathcal{G}(P)$ d'après le lemme 3.1.7
Or $\{s\} \in \mathcal{G}(\{s\})$
Donc $\{s\} \in \mathcal{G}(P)$

$\boldsymbol{\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\} \in \mathcal{G}(P)}$

Dans ce cas :
$\boldsymbol{\{}\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\}\boldsymbol{\}} \cup \mathcal{G}(\{s_1\})\{s\} \cup \mathcal{G}(\{s_2\})\{s\} \cup \mathcal{G}(\{s\})$
$= \mathcal{G}(\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\})$
$\subseteq \mathcal{G}(P)$ d'après le lemme 3.1.7
Or $\{s_1; s\} \in \mathcal{G}(\{s_1\})\{s\}$ et $\{s_2; s\} \in \mathcal{G}(\{s_2\})\{s\}$
Donc $\{s_1; s\}$ et $\{s_2; s\} \in \mathcal{G}(P)$

$\boldsymbol{\{\texttt{while } F\ \{s_1\}; s\} \in \mathcal{G}(P)}$

Dans ce cas :
$\mathcal{G}(\{s_1\})\{\texttt{while } F\ \{s_1\}; s\} \cup \mathcal{G}(\{s\})$
$= \mathcal{G}(\{\texttt{while } F\ \{s_1\}; s\})$
$\subseteq \mathcal{G}(P)$ d'après le lemme 3.1.7
Or $\{s_1; \texttt{while } F\ \{s_1\}; s\} \in \mathcal{G}(\{s_1\})\{\texttt{while } F\ \{s_1\}; s\}$
Et $\{s\} \in \mathcal{G}(\{s\})$
Donc $\{s_1; \texttt{while } F\ \{s_1\}; s\}$ et $\{s\} \in \mathcal{G}(P)$

$\boldsymbol{\{\texttt{loop } n \texttt{ except } F\ \{s_1\}; s\} \in \mathcal{G}(P)}$

De même.

□

Remarque. Dans le `Loop` de Meyer et Ritchie [MR67] la variable n bornant une commande $\texttt{loop } n\ \{s\}$ peut être utilisée dans le corps de la boucle s, mais le nombre d'itérations de la boucle lui est fixé dès la première exécution.

Par exemple le programme $\texttt{loop } n\ \{n := n + 1\}$ a comme effet de doubler la valeur de n.

Pour implémenter ce comportement, la sémantique opérationnelle peut être définie comme dans [APV10] par :

$$\begin{array}{ll} \{\texttt{loop } n\ \{s_1\}; s_2\} \star X \succ \{s_1; \texttt{loop } S^{\overline{n}^X - 1}\emptyset\ \{s_1\}; s_2\} \star X & \text{si } \overline{n}^X > 0 \\ \{\texttt{loop } n\ \{s_1\}; s_2\} \star X \succ \{s_2\} \star X & \text{si } \overline{n}^X = 0 \end{array}$$

Nous n'avons pas pu le faire en raison des graphes d'exécution : il y aurait a priori une quantité non bornée de `loop` $S^n\emptyset$ $\{s_1\}$ qui pourraient apparaître durant une exécution.

Il aurait été possible de faire décroître la valeur de n au fil des itérations, mais ce n'est pas la sémantique habituelle. C'est pour cela que nous avons utilisé un compteur i pour chaque boucle, ce qui correspond à l'usage.

3.2 Traduction d'un programme impératif

Maintenant que nous pouvons indicer un nombre borné de booléens frais par les différents états possibles du programme, nous pouvons définir la traduction d'un programme impératif dans le cadre des ASMs. Chaque état du programme nous renseigne sur la commande de tête, et nous utilisons des conditionnelles pour nous renseigner sur l'état en cours. Ainsi, il est possible de simuler la sémantique opérationnelle suivie par le programme.

Définition de la traduction

Notation. Nous utilisons donc comme variables fraîches les $\{b_{P_j} \mid P_j \in \mathcal{G}(P)\}$.

Une seule b_{P_j} sera vraie pour chaque étape de l'exécution, donc par la suite nous noterons $X[b_{P_j}]$ si b_{P_j} est vraie et que les autres b_{P_k} sont fausses, où X est une $\mathcal{L}(P)$-structure. Ainsi en particulier nous avons $X[b_{P_j}]|_{\mathcal{L}(P)} = X$.

Pour passer d'un état P_j à un état P_k du programme il nous suffira d'exécuter l'ensemble de mises à jour $\{(b_{P_j}, \overline{false}), (b_{P_k}, \overline{true})\}$.

Pour que cet ensemble soit consistant il faut cependant s'assurer qu'il n'existe pas de programme tel que $\tau^1_X(P) = P$. D'après le système de transition 2.2.4 p.47 cela est malheureusement possible.

En effet, si le corps d'une boucle est vide ($s_1 = \epsilon$) alors nous avons :

$\{\texttt{while}\ F\ \{s_1\}; s\} = \{s_1; \texttt{while}\ F\ \{s_1\}; s\}$, et de même pour la boucle `loop`.

Dans ces deux cas nous ferons une distinction dans la traduction pour éviter de rendre inconsistant $\{(b_{P_j}, \mathit{false}), (b_{P_k}, \mathit{true})\}$, tout simplement en ne modifiant pas la b_{P_j} en cours. Toutefois comme l'état n'est alors plus assuré de changer il faut vérifier dans les deux cas si l'ASM va s'arrêter ou non :

1. Dans le cas du `loop`, heureusement, la mise à jour du compteur (rappelons en passant qu'ils font partie du langage, car nous les utiliserons dans la traduction) permet d'empêcher que l'ASM ne s'arrête.
2. Dans le cas du `while` l'ASM s'arrêterait, contrairement au programme impératif qui lui continurait indéfiniment. Pour que la simulation soit correcte, nous introduisons une nouvelle variable booléenne b_∞ initialisée à *false*. Cela nous permettra dans la traduction de faire en boucle la mise à jour $b_\infty := \neg b_\infty$ pour que b_∞ alterne entre *true* et *false* et maintienne artificiellement l'exécution de l'ASM.

La proposition 3.1.8 assure que la traduction suivante est bien définie :

Définition 3.2.1. (Traduction d'un programme impératif par une ASM)

$\Pi_P =_{def} \texttt{par}_{P_j \in \mathcal{G}(P)}\ \texttt{if}\ b_{P_j}\ \texttt{then}\ P_j^{tr}\ \texttt{endpar}$, où :

Cas sans boucle :

$$
\begin{aligned}
\{\}^{tr} &=_{def} \texttt{par endpar} \\
\{ft_1 \ldots t_k := t_0; s\}^{tr} &=_{def} \texttt{par } b_{\{ft_1 \ldots t_k := t_0; s\}} := \mathit{false} \\
&\quad \| \; ft_1 \ldots t_k := t_0 \\
&\quad \| \; b_{\{s\}} := \mathit{true} \\
&\quad \texttt{endpar} \\
\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\}^{tr} &=_{def} \texttt{par } b_{\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\}} := \mathit{false} \\
&\quad \| \; \texttt{if } F \texttt{ then } b_{\{s_1; s\}} := \mathit{true} \\
&\quad \texttt{else } b_{\{s_2; s\}} := \mathit{true} \\
&\quad \texttt{endif} \\
&\quad \texttt{endpar}
\end{aligned}
$$

Si le corps de la boucle est vide alors :

$$
\begin{aligned}
\{\texttt{while } F \; \{\}; s\}^{tr} &=_{def} \texttt{if } F \texttt{ then } b_\infty := \neg b_\infty \\
&\quad \texttt{else} \\
&\quad \texttt{par } b_{\{\texttt{while } F \; \{\}; s\}} := \mathit{false} \\
&\quad \| \; b_{\{s\}} := \mathit{true} \\
&\quad \texttt{endpar} \\
&\quad \texttt{endif} \\
\{\texttt{loop } n \texttt{ except } F \; \{\}; s\}^{tr} &=_{def} \texttt{if } (i \neq n \wedge \neg F) \texttt{ then } i := i + 1 \\
&\quad \texttt{else} \\
&\quad \texttt{par } b_{\{\texttt{loop } n \texttt{ except } F \; \{\}; s\}} := \mathit{false} \\
&\quad \| \; i := 0 \\
&\quad \| \; b_{\{s\}} := \mathit{true} \\
&\quad \texttt{endpar} \\
&\quad \texttt{endif}
\end{aligned}
$$

Si le corps de la boucle n'est pas vide alors :

$$
\begin{aligned}
\{\texttt{while } F \; \{s_1\}; s\}^{tr} &=_{def} \texttt{par } b_{\{\texttt{while } F \; \{s_1\}; s\}} := \mathit{false} \\
&\quad \| \; \texttt{if } F \texttt{ then } b_{\{s_1; \texttt{while } F \; \{s_1\}; s\}} := \mathit{true} \\
&\quad \texttt{else } b_{\{s\}} := \mathit{true} \\
&\quad \texttt{endif} \\
&\quad \texttt{endpar} \\
\{\texttt{loop } n \texttt{ except } F \; \{s_1\}; s\}^{tr} &=_{def} \texttt{par } b_{\{\texttt{loop } n \texttt{ except } F \; \{s_1\}; s\}} := \mathit{false} \\
&\quad \| \; \texttt{if } (i \neq n \wedge \neg F) \texttt{ then} \\
&\quad\quad \texttt{par } i := i + 1 \\
&\quad\quad \| \; b_{\{s_1; \texttt{loop } n \texttt{ except } F \; \{s_1\}; s\}} := \mathit{true} \\
&\quad\quad \texttt{endpar} \\
&\quad \texttt{else} \\
&\quad\quad \texttt{par } i := 0 \\
&\quad\quad \| \; b_{\{s\}} := \mathit{true} \\
&\quad\quad \texttt{endpar} \\
&\quad \texttt{endif} \\
&\quad \texttt{endpar}
\end{aligned}
$$

Exemple 3.2.2. Rappelons l'un des programmes impératifs donné p.44 pour l'algorithme *Min* p.24 :

$$P_{min} = \{r := 0; \texttt{loop } n \texttt{ except } r = m \; \{r := r + 1; \}; \}$$

Son graphe d'exécution a été donné p.62 :

$$\mathcal{G}(P_{min}) = \mathbf{\{} \begin{array}{l} \{r := 0; \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r + 1;\};\}, \\ \{\texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r + 1;\};\}, \\ \{r := r + 1; \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r + 1;\};\}, \\ \{\} \end{array} \mathbf{\}}$$

La traduction $\Pi_{P_{min}}$ de ce programme en ASM [1] est à la table 3.4 p.67.

TABLE 3.4 – Traduction de P_{min} en ASM

```
if b_{r:=0;loop n except r=m {r:=r+1;};} then
   b_{r:=0;loop n except r=m {r:=r+1;};} := false
   r := 0
   b_{loop n except r=m {r:=r+1;};} := true
if b_{loop n except r=m {r:=r+1;};} then
   b_{loop n except r=m {r:=r+1;};} := false
   if (i ≠ n ∧ r ≠ m) then
      i := i + 1
      b_{r:=r+1;loop n except r=m {r:=r+1;};} := true
   else
      i := 0
      b_{{}} := true
if b_{r:=r+1;loop n except r=m {r:=r+1;};} then
   b_{r:=r+1;loop n except r=m {r:=r+1;};} := false
   r := r + 1
   b_{loop n except r=m {r:=r+1;};} := true
if b_{{}} then
```

Bien sûr la dernière conditionnelle pourrait être enlevée, mais nous avons préféré faire une présentation homogène.

Par soucis de lisibilité, nous ne noterons pas dans le résultat suivant les changements de valeur de b_∞, mais le cas sera bien abordé dans la preuve et par la suite :

Proposition 3.2.3. *(Simulation étape par étape)*
Pour tout $0 \leq i < time(P, X)$: $\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}]) = \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

Démonstration. $i < time(P, X)$ donc $\tau_X^i(P) \neq \{\}$
La preuve se fait par cas sur $\tau_X^i(P)$:

$\boldsymbol{\tau_X^i(P) = \{u; s\}}$

Dans ce cas, d'une part :
$\tau_P^{i+1}(X) = \tau_P^i(X) \oplus \overline{u}^{\tau_P^i(X)}$ et $\tau_X^{i+1}(P) = \{s\}$
Et d'autre part :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\{u; s\}^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$

1. ↑ Pour simplifier les notations, les commandes `par`...`endpar` ont été rendues implicites par l'indentation.

$= \Delta(\texttt{par}\ b_{\{u;s\}} := \mathit{false} \parallel u \parallel b_{\{s\}} := \mathit{true}\ \texttt{endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{u;s\}}, \mathit{false}), \overline{u}^{\tau_P^i(X)[b_{\tau_X^i(P)}]}, (b_{\{s\}}, \mathit{true})\}$
$= \{(b_{\{u;s\}}, \mathit{false}), \overline{u}^{\tau_P^i(X)}, (b_{\{s\}}, \mathit{true})\}$ (car $b_{\tau_X^i(P)}$ est fraîche)
Donc :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\{u;s\}}] \oplus \{(b_{\{u;s\}}, \mathit{false}), \overline{u}^{\tau_P^i(X)}, (b_{\{s\}}, \mathit{true})\}$
$= (\tau_P^i(X) \oplus \overline{u}^{\tau_P^i(X)})[b_{\{s\}}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

$\boldsymbol{\tau_X^i(P) = \{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}}$

Dans ce cas, d'une part :
$\tau_P^{i+1}(X) = \tau_P^i(X)$ et $\tau_X^{i+1}(P) = \{s_i; s\}$
où $i = 1$ si F est vraie dans $\tau_P^i(X)$, et sinon $i = 2$
Et d'autre part :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\texttt{par}\ b_{\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\};s\}} := \mathit{false}$
$\parallel \texttt{if}\ F\ \texttt{then}\ b_{\{s_1;s\}} := \mathit{true}\ \texttt{else}\ b_{\{s_2;s\}} := \mathit{true}\ \texttt{endif}$
$\texttt{endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\};s\}}, \mathit{false}), (b_{\{s_j;s\}}, \mathit{true})\}$
où $j = 1$ si F est vraie dans $\tau_P^i(X)[b_{\tau_X^i(P)}]$, et sinon $j = 2$
Or $b_{\tau_X^i(P)}$ est fraîche, donc $\overline{F}^{\tau_P^i(X)} = \overline{F}^{\tau_P^i(X)[b_{\tau_X^i(P)}]}$, d'où $i = j$
Donc :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\};s\}}, \mathit{false}), (b_{\{s_i;s\}}, \mathit{true})\}$
D'où :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\};s\}}]$
$\oplus\{(b_{\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\};s\}}, \mathit{false}), (b_{\{s_i;s\}}, \mathit{true})\}$
$= \tau_P^i(X)[b_{\{s_i;s\}}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

$\boldsymbol{\tau_X^i(P) = \{\texttt{while}\ F\ \{s_1\}; s\}}$

Dans ce cas $\tau_P^{i+1}(X) = \tau_P^i(X)$

$\boldsymbol{s_1 = \epsilon}$

Dans ce cas, d'une part :
$\tau_X^{i+1}(P) = \{\texttt{while}\ F\ \{\}; s\} = \tau_X^i(P)$ si F est vraie dans $\tau_P^i(X)$,
et sinon $\tau_X^{i+1}(P) = \{s\}$
Et d'autre part :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\{\texttt{while}\ F\ \{\}; s\}^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$

$= \Delta(\texttt{if } F \texttt{ then } b_\infty := \neg b_\infty$
$\texttt{else par } b_{\{\texttt{while } F\ \{\};s\}} := \mathit{false} \parallel b_{\{s\}} := \mathit{true} \texttt{ endpar}$
$\texttt{endif}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= (b_\infty, \overline{\neg b_\infty}^{\tau_P^i(X)[b_{\tau_X^i(P)}]})$ si F est vraie dans $\tau_P^i(X)$,
et $\{(b_{\{\texttt{while } F\ \{\};s\}}, \mathit{false}), (b_{\{s\}}, \mathit{true})\}$ sinon

F est vraie dans $\tau_P^i(X)$

Dans ce cas :
$\tau_X^{i+1}(P) \star \tau_P^{i+1}(X) = \tau_X^i(P) \star \tau_P^i(X)$
Donc l'exécution est infinie.
Comme $\overline{F}^{\tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]} = \overline{F}^{\tau_P^i(X)[b_{\tau_X^i(P)}]}$ l'ASM reviendra toujours au même cas, et elle fera en boucle la mise à jour $(b_\infty, \overline{\neg b_\infty}^{\tau_P^i(X)[b_{\tau_X^i(P)}]})$.
Ainsi la valeur de b_∞ va alterner entre *true* et *false* à chaque étape, empêchant l'ASM de s'arrêter.
Les autres symboles sont inchangés, donc on a bien :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}]) = \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

F est fausse dans $\tau_P^i(X)$

Dans ce cas :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{while } F\ \{\};s\}}, \mathit{false}), (b_{\{s\}}, \mathit{true})\}$
Donc :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\{\texttt{while } F\ \{\};s\}}] \oplus \{(b_{\{\texttt{while } F\ \{\};s\}}, \mathit{false}), (b_{\{s\}}, \mathit{true})\}$
$= \tau_P^i(X)[b_{\{s\}}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

$s_1 \neq \epsilon$

Dans ce cas, d'une part :
$\tau_X^{i+1}(P) = \{s_1; \texttt{while } F\ \{s_1\}; s\}$ si F est vraie dans $\tau_P^i(X)$,
et sinon $\tau_X^{i+1}(P) = \{s\}$
Et d'autre part :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\{\texttt{while } F\ \{s_1\}; s\}^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\texttt{par } b_{\{\texttt{while } F\ \{s_1\};s\}} := \mathit{false}$
$\parallel \texttt{if } F \texttt{ then } b_{\{s_1;\texttt{while } F\ \{s_1\};s\}} := \mathit{true} \texttt{ else } b_{\{s\}} := \mathit{true} \texttt{ endif}$
$\texttt{endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{while } F\ \{s_1\};s\}}, \mathit{false}), (b_Q, \mathit{true})\}$
où $Q = \{s_1; \texttt{while } F\ \{s_1\}; s\}$ si F est vraie dans $\tau_P^i(X)[b_{\tau_X^i(P)}]$,
et sinon $Q = \{s\}$
Or $b_{\tau_X^i(P)}$ est fraîche
Donc $\overline{F}^{\tau_P^i(X)} = \overline{F}^{\tau_P^i(X)[b_{\tau_X^i(P)}]}$
D'où $Q = \tau_X^{i+1}(P)$, et :

$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{while } F \ \{s_1\};s\}}, \mathit{false}), (b_{\tau_X^{i+1}(P)}, \mathit{true})\}$
D'où :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\{\texttt{while } F \ \{s_1\};s\}}] \oplus \{(b_{\{\texttt{while } F \ \{s_1\};s\}}, \mathit{false}), (b_{\tau_X^{i+1}(P)}, \mathit{true})\}$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

$\boldsymbol{\tau_X^i(P) = \{\texttt{loop } n \texttt{ except } F \ \{s_1\}; s\}}$

Deux cas :

$\boldsymbol{s_1 = \epsilon}$

D'une part, selon les cas :

$i < n$ et F est fausse dans $\tau_P^i(X)$

Dans ce cas :
$\tau_X^{i+1}(P) = \{\texttt{loop } n \texttt{ except } F \ \{\}; s\} = \tau_X^i(P)$
et $\tau_P^{i+1}(X) = \tau_X^i(P) \oplus (i, \bar{i}^{\tau_P^i(X)} + 1)$

$i = n$ ou F est vraie dans $\tau_P^i(X)$

Dans ce cas :
$\tau_X^{i+1}(P) = \{s\}$
et $\tau_P^{i+1}(X) = \tau_X^i(P) \oplus (i, 0)$

Et d'autre part :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\{\texttt{loop } n \texttt{ except } F \ \{\}; s\}^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\texttt{if } (i \neq n \wedge \neg F) \texttt{ then } i := i + 1$
$\texttt{else par } b_{\{\texttt{loop } n \texttt{ except } F \ \{\};s\}} := \mathit{false} \parallel i := 0 \parallel b_{\{s\}} := \mathit{true} \texttt{ endpar}$
$\texttt{endif}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
Donc, selon les cas :

$i < n$ et F est fausse dans $\tau_P^i(X)$

Dans ce cas :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= (i, \bar{i}^{\tau_P^i(X)} + 1)$
Donc :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus (i, \bar{i}^{\tau_P^i(X)} + 1)$
$= (\tau_P^i(X) \oplus (i, \bar{i}^{\tau_P^i(X)} + 1))[b_{\tau_X^i(P)}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

$i = n$ ou F est vraie dans $\tau_P^i(X)$

Dans ce cas :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{loop } n \texttt{ except } F \ \{\};s\}}, \mathit{false}), (i, 0), (b_{\{s\}}, \mathit{true})\}$
Donc :

$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{\};s\}}]$
$\oplus\{(b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{\};s\}}, \mathit{false}), (i,0), (b_{\{s\}}, \mathit{true})\}$
$= (\tau_P^i(X) \oplus (i,0))[b_{\{s\}}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

$\boldsymbol{s_1 \neq \epsilon}$

D'une part, selon les cas :

$i < n$ et F est fausse dans $\tau_P^i(X)$

Dans ce cas :
$\tau_X^{i+1}(P) = \{s_1; \texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s\}$
et $\tau_P^{i+1}(X) = \tau_X^i(P) \oplus (i, \bar{i}^{\tau_P^i(X)} + 1)$

$i = n$ ou F est vraie dans $\tau_P^i(X)$

Dans ce cas :
$\tau_X^{i+1}(P) = \{s\}$
et $\tau_P^{i+1}(X) = \tau_X^i(P) \oplus (i,0)$

Et d'autre part :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s\}^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \Delta(\texttt{par}\ b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}} := \mathit{false}$
$\|\ \texttt{if}\ (i \neq n \wedge \neg F)\ \texttt{then}$
$\texttt{par}\ i := i+1\ \|\ b_{\{s_1;\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}} := \mathit{true}\ \texttt{endpar}$
$\texttt{else par}\ i := 0\ \|\ b_{\{s\}} := \mathit{true}\ \texttt{endpar endif}$
$\texttt{endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$

Donc, selon les cas :

$i < n$ et F est fausse dans $\tau_P^i(X)$

Dans ce cas :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}, \mathit{false}), (i, \bar{i}^{\tau_P^i(X)} + 1),$
$(b_{\{s_1;\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}, \mathit{true})\}$

Donc :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}]$
$\oplus\ \{(b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}, \mathit{false}),$
$(i, \bar{i}^{\tau_P^i(X)} + 1), (b_{\{s_1;\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}, \mathit{true})\}$
$= (\tau_P^i(X) \oplus (i, \bar{i}^{\tau_P^i(X)} + 1))[b_{\{s_1;\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

$i = n$ ou F est vraie dans $\tau_P^i(X)$

Dans ce cas :
$\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}, \mathit{false}), (i,0), (b_{\{s\}}, \mathit{true})\}$

Donc :
$\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] \oplus \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}]$
$\quad \oplus \{(b_{\{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\};s\}}, \mathit{false}), (i,0), (b_{\{s\}}, \mathit{true})\}$
$= (\tau_P^i(X) \oplus (i,0))[b_{\{s\}}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

□

La simulation

Théorème 3.2.4. `ASM` *simule* `Imp`

Démonstration. Rappelons que la définition de la simulation «raisonnable » p.56 nécessite trois conditions :

1. $\mathcal{L}(\Pi_P) = \mathcal{L}(P) \cup \{b_{P_j} \mid P_j \in \mathcal{G}(P)\} \cup \{b_\infty\}$
   où $card(\{b_{P_j} \mid P_j \in \mathcal{G}(P)\}) \leq length(P) + 1$ d'après le lemme 3.1.5.
   Donc au pire nous utilisons $\boxed{length(P) + 2 \text{ variables temporaires}}$
2. Prouvons[1] par récurrence sur i que pour tout $0 \leq i \leq time(P, X)$:
   $$\tau_{\Pi_P}^i(X[b_P]) = \tau_P^i(X)[b_{\tau_X^i(P)}]$$
   $\boldsymbol{i = 0}$
   $\tau_{\Pi_P}^0(X[b_P]) = X[b_P] = \tau_P^0(X)[b_{\tau_X^0(P)}]$
   $\boldsymbol{i \Rightarrow i+1}$
   $\tau_{\Pi_P}^{i+1}(X[b_P])$
   $= \tau_{\Pi_P}(\tau_{\Pi_P}^i(X[b_P]))$
   $= \tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$ d'après l'hypothèse d'induction
   $= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$ d'après la proposition 3.2.3
   En prenant $0 \leq i < i+1 \leq time(P, X)$
   Donc pour tout $i \in \mathbb{N}$:
   $$\tau_{\Pi_P}^i(X[b_P])|_{\mathcal{L}(P)} = \tau_P^i(X)[b_{\tau_X^i(P)}]|_{\mathcal{L}(P)} = \tau_P^i(X)$$
   D'où la dilatation temporelle est de $\boxed{d = 1}$
3. Montrons par inégalités que $time(\Pi_P, X) = time(P, X)$:
   $\boldsymbol{time(\Pi_P, X) \leq time(P, X)}$
   Si $i = time(P, X)$ alors $\tau_X^i(P) = \{\}$
   Donc :
   $\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
   $= \Delta(\{\}^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
   $= \Delta(\texttt{par endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
   $= \varnothing$

1. ↑ Ici encore, par soucis de lisibilité, nous ne notons pas la valeur de b_∞, mais le cas est bien abordé dans la preuve.

D'où :
$\tau^{i+1}_{\Pi_P}(X[b_P])$
$= \tau_{\Pi_P}(\tau^i_{\Pi_P}(X[b_P]))$
$= \tau_{\Pi_P}(\tau^i_P(X)[b_{\tau^i_X(P)}])$
$= \tau^i_P(X)[b_{\tau^i_X(P)}] + \Delta(\Pi_P, \tau^i_P(X)[b_{\tau^i_X(P)}])$
$= \tau^i_P(X)[b_{\tau^i_X(P)}]$
$= \tau^i_{\Pi_P}(X[b_P])$
Et nous avons bien $time(\Pi_P, X) \leq time(P, X)$

$\boldsymbol{time(\Pi_P, X) \geq time(P, X)}$
La preuve se fait par cas pour tout $0 \leq i < time(P, X)$:

$\boldsymbol{\tau^{i+1}_X(P) \neq \tau^i_X(P)}$
Dans ce cas $\tau^{i+1}_P(X)[b_{\tau^{i+1}_X(P)}] \neq \tau^i_P(X)[b_{\tau^i_X(P)}]$.
Donc $\tau^{i+1}_{\Pi_P}(X[b_P]) \neq \tau^i_{\Pi_P}(X[b_P])$.

$\boldsymbol{\tau^{i+1}_X(P) = \tau^i_X(P)}$
Alors d'après la sémantique opérationnelle de `Imp` p.47 :

$\boldsymbol{\tau^i_X(P) = \{\texttt{while } F \ \{\}; s\}}$
avec F vraie dans $\tau^i_P(X)$
Dans ce cas, comme vu dans la preuve de la proposition 3.2.3 :

$$\tau^{i+1}_P(X)[b_{\tau^{i+1}_X(P)}] = \tau^i_P(X)[b_{\tau^i_X(P)}]$$

Mais b_∞ change de valeur grâce à $b_\infty := \neg b_\infty$.
Donc en fait $\tau^{i+1}_{\Pi_P}(X[b_P]) \neq \tau^i_{\Pi_P}(X[b_P])$.

$\boldsymbol{\tau^i_X(P) = \{\texttt{loop } n \texttt{ except } F \ \{\}; s\}}$
avec $i < n$ et F fausse dans $\tau^i_P(X)$
Dans ce cas $\tau^{i+1}_P(X) = \tau^i_X(P) \oplus (i, \bar{i}^{\tau^i_P(X)} + 1)$
D'où $\tau^{i+1}_P(X)[b_{\tau^{i+1}_X(P)}] \neq \tau^i_P(X)[b_{\tau^i_X(P)}]$.
Donc $\tau^{i+1}_{\Pi_P}(X[b_P]) \neq \tau^i_{\Pi_P}(X[b_P])$.

Dans tous les cas, nous avons $\tau^{i+1}_{\Pi_P}(X[b_P]) \neq \tau^i_{\Pi_P}(X[b_P])$.
Donc $time(\Pi_P, X) \geq time(P, X)$.

Nous avons donc $time(\Pi_P, X) = time(P, X)$, d'où $\boxed{e = 0}$

□

Remarque. La traduction p.65 peut être vue comme une implémentation de la sémantique opérationnelle des programmes impératifs, qui illustre la puissance du modèle des ASMs créé par Yuri Gurevich.

Dans cet ordre d'idée, Serge Grigorieff et Pierre Valarcher ont travaillé dans [GV10] sur l'utilisation des ASMs comme modèle unificateur des modèles de calcul en général.

Ainsi, les ASMs pourraient être vues non comme un modèle de calcul avec ses programmes mais comme une modélisation des processeurs eux-mêmes.

3.3 Temps primitif récursif

Ainsi, les ASMs peuvent simuler nos programmes impératifs, et de plus elles le font strictement pas-à-pas au sens de p.55. Nous allons maintenant raffiner un peu cette simulation.

Soit $\texttt{Imp}_{\mathcal{C}}$ une partie des programmes impératifs telle que pour tout $P \in \texttt{Imp}_{\mathcal{C}}$ la complexité de P est $c_P \in \mathcal{C}$. Dans ce cas, comme ASM simule Imp nous aurons que $\texttt{ASM}_{\mathcal{C}}$ simule $\texttt{Imp}_{\mathcal{C}}$, au sens de la classe en temps définie p.33 (et en rappelant que $\texttt{Algo} = \texttt{ASM}$).

Dans la fin du chapitre nous nous contenterons donc d'établir des classes de temps pour des fragments pertinents $\texttt{Imp}_{\mathcal{C}}$ de Imp. En particulier nous montrerons comme annoncé plus tôt que :

1. LoopC a un temps $\mathcal{PR}$, si l'espace est également $\mathcal{PR}$.
2. LoopC a un temps $\mathcal{P}ol$, si les bornes des boucles sont statiques.

Complexité en espace

Dans cette section nous montrons que LoopC est (sans surprise) primitif récursif en temps. Toutefois ce n'est pas une simple répétition de [MR67] ou [APV10] puisque nous généralisons le résultat à toutes les structures de données primitives récursives, et pas seulement aux booléens ou aux entiers unaires.

Pourquoi cette restriction sur l'espace ? Une commande `loop` n transforme une valeur $\overline{n}^X$ en temps d'exécution donc il est raisonnable d'exiger que les opérations ne fassent pas trop monter la taille des éléments :

Exemple 3.3.1. (Un programme de LoopC qui n'est pas $\mathcal{PR}$-time)

Le programme $\{n := ackermann(n, n); \texttt{loop}\ n\ \{\}; \}$ est dans LoopC, mais il n'est pas en temps primitif récursif malgré le fait qu'il n'y ait pas de `while`.

Ainsi, il faut des restrictions sur les opérations utilisables.

Définition 3.3.2. (Classe en espace)

L'algorithme A est $\mathcal{C}$-space si pour toute opération $f \in Oper(A)$ il existe $\varphi_f \in \mathcal{C}$ telle que pour tout état X et pour tout α_f-uplet $\vec{a}$ d'éléments de $\mathcal{U}(X)$:

$$|\overline{f}^X(\vec{a})| \leq \varphi_f(|\vec{a}|)$$

Remarque. Nous prenons $f \in Oper(A)$ et pas $f \in Cons(A) \sqcup Oper(A)$ car nous utilisons la remarque p.142 dans le cadre de notre présentation des états représentables : si f est un constructeur alors $|\overline{f}^X(\overline{t_1}^X, \ldots, \overline{t_k}^X)| = 1 + \sum_{i=1}^{k} |\overline{t_i}^X|$. Toutefois si les états sont mesurables (au sens de la p.30) et non représentables, il faut bien utiliser $f \in Cons(A) \sqcup Oper(A)$ dans la définition précédente.

Par la suite nous dirons qu'une fonction $\varphi \in \mathcal{C}$ est **monotone** si pour tout $\vec{m} \leq \vec{n}$ (c'est-à-dire que pour tout i : $m_i \leq n_i$) nous avons $\varphi(\vec{m}) \leq \varphi(\vec{n})$. Nous avons besoin que les fonctions $\varphi \in \mathcal{C}$ soient monotones afin de pouvoir établir des bornes en temps, mais heureusement le lemme suivant permet de toujours supposer que c'est bien le cas dans le cas $\mathcal{PR}$:

Lemme 3.3.3. *Pour toute $\varphi \in \mathcal{PR}$ il existe $\psi \in \mathcal{PR}$ monotone telle que pour tout $\vec{n}$: $\varphi(\vec{n}) \leq \psi(\vec{n})$*

Démonstration. Selon l'arité de φ :

φ est d'arité 0

Dans ce cas elle est monotone.

φ est d'arité 1

Dans ce cas la preuve est faite par induction sur n :

$n = 0$

Il faut prendre $\psi(0) = \varphi(0)$

$n \Rightarrow n + 1$

Dans ce cas il faut prendre $\psi(n+1) = max(\varphi(n+1), \psi(n))$

Ainsi nous avons $\psi(n+1) \geq \varphi(n+1)$

Et pour la monotonie, soit $0 \leq i \leq n$:

$\psi(n+1) \geq \psi(n) \geq \psi(i)$ d'après l'hypothèse d'induction

Ainsi la fonction suivante convient :

$$\psi(n) = max(\varphi(n), \texttt{if } n = 0 \texttt{ then } 0 \texttt{ else } \psi(n-1))$$

φ est d'arité $\alpha > 1$

Généralisons l'idée précédente avec la fonction :

$$\psi(\vec{n}) = max(\varphi(\vec{n}), max_{1 \leq i \leq \alpha}(\texttt{if } n_i = 0 \texttt{ then } 0 \texttt{ else } \psi(n_1, \ldots n_i - 1 \ldots n_\alpha)))$$

La preuve se fait par récurrence sur $N = \sum_{1 \leq i \leq \alpha} n_i$:

$N = 0$

Dans ce cas $\vec{n} = \vec{0}$, donc $\psi(\vec{0}) = \varphi(\vec{0})$

$N \Rightarrow N + 1$

Dans ce cas nous avons $\psi(\vec{n}) \geq \varphi(\vec{n})$ par construction.

Les $\vec{m} \leq \vec{n}$ sans être $\vec{n}$ sont tels que $\sum_{1 \leq i \leq \alpha_\varphi} m_i < \sum_{1 \leq i \leq \alpha_\varphi} n_i$.

Si $\sum_{1 \leq i \leq \alpha_\varphi} n_i = N + 1$ alors les $\vec{m}$ tels que $\sum_{1 \leq i \leq \alpha_\varphi} m_i = N$ sont les $\vec{n}$ où on a enlevé 1 à une coordonnée non nulle.

Donc par construction $\psi(\vec{n}) \geq \psi(\vec{m})$, ce qui par induction montre la monotonie.

□

Rappel. de la définition p.32 :

$$|X| = (|f|_X)_{f \in Dyn(A) \sqcup Init(A)}, \text{ où } |f|_X = \sup_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$$

Si l'espace est $\mathcal{PR}$-space, alors nous montrons que la taille des termes est bornée par une fonction $\mathcal{PR}$ de la taille de l'état en cours. Ainsi, le lemme suivant permet de borner l'accroissement de l'espace en cas de mise à jour :

Lemme 3.3.4. *Si A est $\mathcal{PR}$-space alors pour tout $t \in T(A)$ il existe $\varphi_t \in \mathcal{PR}$ monotone telle que pour tout $X \in S(A)$: $|\overline{t}^X| \leq \varphi_t(|X|)$*

Démonstration. Par induction sur t :

$t = c$

La preuve se fait par cas sur c :

$c \in Dyn(A) \sqcup Init(A)$

Dans ce cas $|\overline{t}^X| = |\overline{c}^X| = |c|_X$

et φ_t est une projection, qui est monotone.

$\boldsymbol{c \in Cons(A)}$

Dans ce cas $|\overline{t}^X| = |\overline{c}^X| = 1$

et $\varphi_t = 1$, qui est monotone.

$\boldsymbol{c \in Oper(A)}$

Dans ce cas[1] par hypothèse il existe $\varphi_t \in \mathbb{N}_1$ tel que $|\overline{c}| \leq \varphi_t$.

De plus φ_t est constante donc monotone.

$\boldsymbol{t = ft_1 \dots t_k}$ **avec** $\boldsymbol{k > 0}$

Dans ce cas $|\overline{t}^X| = |\overline{ft_1 \dots t_k}^X| = |\overline{f}^X(\overline{t_1}^X, \dots, \overline{t_k}^X)|$

De plus, $T(A)$ est supposé clos par sous-termes, donc $t_1, \dots t_k \in T(A)$ et nous pouvons leur appliquer l'hypothèse d'induction.

La preuve se fait par cas sur f :

$\boldsymbol{f \in Dyn(A) \sqcup Init(A)}$

$$\begin{aligned} |\overline{t}^X| &= |\overline{f}^X(\overline{t_1}^X, \dots, \overline{t_k}^X)| \\ &\leq \sup_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})| \\ &= |f|_X \end{aligned}$$

et φ_t est une projection, qui est monotone.

$\boldsymbol{f \in Cons(A)}$

$$\begin{aligned} |\overline{t}^X| &= |\overline{f}^X(\overline{t_1}^X, \dots, \overline{t_k}^X)| \\ &= 1 + \textstyle\sum_{i=1}^k |\overline{t_i}^X| \text{ (voir la remarque p.142)} \\ &\leq 1 + \textstyle\sum_{i=1}^k \varphi_{t_i}(|X|) \text{ (par induction)} \end{aligned}$$

et $\varphi_t = 1 + \sum_{i=1}^k \varphi_{t_i}$.

De plus comme la somme est monotone φ_t l'est aussi.

$\boldsymbol{f \in Oper(A)}$

Alors par hypothèse il existe $\varphi_f \in \mathcal{PR}$ telle que pour tout α_f-uplet $\vec{a}$ d'éléments de $\mathcal{U}(A)$: $|\overline{f}(\vec{a})| \leq \varphi_f(|\vec{a}|)$, d'où :

$$\begin{aligned} |\overline{t}^X| &= |\overline{f}^X(\overline{t_1}^X, \dots, \overline{t_k}^X)| \\ &\leq \varphi_f(|\overline{t_1}^X|, \dots |\overline{t_k}^X|) \end{aligned}$$

Or par induction : $|\overline{t_i}^X| \leq \varphi_{t_i}(|X|)$

D'où par monotonie de φ_f (lemme 3.3.3) :

$$|\overline{t}^X| \leq \varphi_f(\varphi_{t_1}(|X|), \dots \varphi_{t_k}(|X|))$$

et $\varphi_t = \varphi_f(\varphi_{t_1}, \dots \varphi_{t_k})$ convient.

De plus φ_t est la composition de fonctions monotones donc est monotone aussi.

□

1. ↑ Une opération d'arité 0 peut être vue comme un paramètre uniforme.

Un fragment primitif récursif

Rappel. Les programmes de `LoopC` sont terminaux (voir le corollaire p.50).

Notation. Soit $Dyn(P) \sqcup Init(P) = \{f_1, \ldots, f_k\}$.
Dans la proposition suivante nous noterons[1] $\vec{\psi}(|X|)$ le k-uplet
$(\psi_1(|f_1|_X, \ldots, |f_k|_X), \ldots, \psi_k(|f_1|_X, \ldots, |f_k|_X))$
et $|X| \leq |Y|$ si pour tout $1 \leq j \leq k$: $|f_j|_X \leq |f_j|_Y$.

Proposition 3.3.5. *(Temps primitif récursif)*
Pour tout $P \in$ `LoopC`, si les états sont $\mathcal{PR}$-space alors il existe $\vec{\psi}^P, \varphi^P \in \mathcal{PR}$ monotones telles que pour tout X :

1. $|P(X)| \leq \vec{\psi}^P(|X|)$
2. $time(P, X) \leq \varphi^P(|X|)$

Démonstration. D'après le lemme 3.3.3, par la suite toutes les fonctions $\mathcal{PR}$ (utilisées comme majorant uniquement) seront considérées monotones.

La preuve se fait par induction sur P :

$\boldsymbol{P = \{\}}$
$\{\}(X) = X$ donc $|P(X)| = |X|$
D'où $\psi_j^P(x_1, \ldots, x_k) = x_j$ convient.
De plus $time(\{\}, X) = 0$ donc $\varphi^P = 0$ convient.

$\boldsymbol{P = P_1 P_2}$
Montrons que si la proposition est vraie pour P_1 et P_2 elle est vraie aussi pour $P = P_1 P_2$:
D'après la proposition 2.2.9 :

1. $P_1 P_2(X) = P_2(P_1(X))$, donc :
   $|P_1 P_2(X)| = |P_2(P_1(X))| \leq \vec{\psi}^{P_2}(|P_1(X)|) \leq \vec{\psi}^{P_2}(\vec{\psi}^{P_1}(|X|))$
   D'où $\psi_j^P = \psi_j^{P_2}(\vec{\psi}^{P_1})$ convient.
2. $time(P_1 P_2, X) = time(P_1, X) + time(P_2, P_1(X))$, donc :
   $time(P_1 P_2, X) \leq \varphi^{P_1}(|X|) + \varphi^{P_2}(|P_1(X)|)$
   $\leq \varphi^{P_1}(|X|) + \varphi^{P_2}(\vec{\psi}^{P_1}(|X|))$
   D'où $\varphi^P = \varphi^{P_1} + \varphi^{P_2}(\vec{\psi}^{P_1})$ convient.

Il ne reste alors plus qu'à prouver la proposition pour les commandes seules en utilisant l'hypothèse d'induction :

$\boldsymbol{P = \{f t_1 \ldots t_k := t_0;\}}$
Si $g \neq f$ ou $(a_1, \ldots, a_k) \neq (\overline{t_1}^X, \ldots, \overline{t_k}^X)$ alors $\overline{g}^{f t_1 \ldots t_k := t_0(X)}(\vec{a}) = \overline{g}^X(\vec{a})$
Le seul changement est que $\overline{f}^{f t_1 \ldots t_k := t_0(X)}(\overline{t_1}^X, \ldots, \overline{t_k}^X) = \overline{t_0}^X$

1. ↑ Si $|X|$ était la somme ou le maximum de la taille des entrées, alors une seule fonction ψ aurait été nécessaire, et nous aurions pu construire φ^P pour qu'elle serve de borne à la fois en espace et en temps. Le résultat est donc plus général mais moins lisible.

Ainsi $|g|_{ft_1\ldots t_k:=t_0(X)} = |g|_X$, et :

$$\begin{aligned}
|f|_{ft_1\ldots t_k:=t_0(X)} &= \sup_{a_i\in\mathcal{U}(A)} |\overline{f}^{ft_1\ldots t_k:=t_0(X)}(\vec{a})| \\
&= max(\sup_{\vec{a}\neq\vec{t^X}} |\overline{f}^{ft_1\ldots t_k:=t_0(X)}(\vec{a})|, |\overline{f}^{ft_1\ldots t_k:=t_0(X)}(\vec{t^X})|) \\
&= max(\sup_{\vec{a}\neq\vec{t^X}} |\overline{f}^{X}(\vec{a})|, |\overline{t_0}^{X}|) \\
&\leq max(\sup_{a_i\in\mathcal{U}(A)} |\overline{f}^{X}(\vec{a})|, |\overline{t_0}^{X}|) \\
&= max(|f|_X, |\overline{t_0}^{X}|)
\end{aligned}$$

D'après le lemme 3.3.4 il existe $\varphi_{t_0} \in \mathcal{PR}$ telle que $|\overline{t_0}^{X}| \leq \varphi_{t_0}(|X|)$
D'où $|f|_{ft_1\ldots t_k:=t_0(X)} \leq max(|f|_X, \varphi_{t_0}(|X|))$
Ainsi $\psi_j^P = max(\pi_j^k, \varphi_{t_0})$ convient, où f est le j-ième symbole.
Et comme $|g|_{ft_1\ldots t_k:=t_0(X)} = |g|_X$, pour les autres une projection convient.
Enfin, comme $time(\{ft_1\ldots t_k := t_0;\}, X) = 1$, $\varphi^P = 1$ convient.

$\boldsymbol{P = \{}$**`if`** $\boldsymbol{F}$ **`then`** $\boldsymbol{P_1}$ **`else`** $\boldsymbol{P_2;\}}$
$P \star X \succ P_i \star X$,
où $i = 1$ si F est vraie dans X et $i = 2$ si F est fausse dans X
D'où $P(X) = P_i(X)$ et $time(P, X) = 1 + time(P_i, X)$
Donc $\psi_j^P = max(\psi_j^{P_1}, \psi_j^{P_2})$, $\varphi^P = 1 + max(\varphi^{P_1}, \varphi^{P_2})$ conviennent.

$\boldsymbol{P = \{}$**`loop`** $\boldsymbol{n}$ **`except`** $\boldsymbol{F}$ $\boldsymbol{P_1;\}}$
En reprenant la preuve du corollaire 2.2.10 p.50 nous avons que :

1. $P(X) = P_1^{\overline{n}^X}(X) \oplus (i, 0)$
2. $time(P, X) = \sum_{0\leq i\leq \overline{n}^X-1}(1 + time(P_1, P_1^i(X))) + 1$

Or, d'après l'hypothèse d'induction :

1. $|P_1(X)| \leq \vec{\psi}^{P_1}(|X|)$
2. $time(P_1, X) \leq \varphi^{P_1}(|X|)$

Donc :

1. Pour tout $1 \leq j \leq k$ nous définissons par récursion mutuelle [1] :
   a. $\psi_j(\vec{x}, 0) = x_j$
   b. $\psi_j(\vec{x}, n + 1) = \psi_j^{P_1}(\psi_1(\vec{x}, n), \ldots, \psi_k(\vec{x}, n))$

   Il faudrait donc prendre $\psi_j^P(|X|) = \psi_j(|X|, \overline{n}^X)$, mais cela nécessite d'abord de vérifier que $\overline{n}^X$ ne dépend que de $|X|$.
   Comme n est interprété dans $\mathbb{N}_1$ p.19, nous avons que $\overline{n}^X = |n|_X - 1$.
   Comme le prédécesseur est primitif récursif, il reste à montrer que $|n|_X$ ne dépend que de $|X|$. Il y a deux cas :
   a. Si $n \in Dyn(P) \sqcup Init(P)$ alors $|n|_X \in |X|$.
   b. Sinon $|n|_X$ est uniforme donc ne dépend pas de l'état en cours.

1. ↑ En étendant l'exemple 1.13 (p.17) de [CL03b] donné pour la récursion double, nous avons que $\mathcal{PR}$ est stable par récursion mutuelle de k fonctions.

Dans tous les cas nous obtenons $|P_1^{\overline{n}^X}(X)| \leq \vec{\psi}^P(|X|)$.
Reste le cas $i \in Dyn(P) \sqcup Init(P)$.
Comme i est initialisé à 0, par la sémantique opérationnelle p.47, nous avons que pour tout état intermédiaire Y de l'exécution de P sur X : $0 \leq \vec{i}^Y \leq \overline{n}^X$.
Ainsi, si i est le j-ième symbole il suffit de prendre :

$$\psi_j^P(|X|) = max(\overline{n}^X, \psi_j(|X|, \overline{n}^X))$$

pour borner i durant l'exécution.
D'où : $|P(X)| \leq \vec{\psi}^P(|X|)$

2. $time(P_1, P_1^i(X)) \leq \varphi^{P_1}(|P_1^i(X)|) \leq \varphi^{P_1}(\vec{\psi}(|X|, i))$, d'où :

$$\begin{aligned} time(P, X) &= \sum_{0 \leq i \leq \overline{n}^X - 1} (1 + time(P_1, P_1^i(X))) + 1 \\ &= 1 + \overline{n}^X + \sum_{0 \leq i \leq \overline{n}^X - 1} time(P_1, P_1^i(X)) \\ &\leq 1 + \overline{n}^X + \sum_{0 \leq i \leq \overline{n}^X - 1} \varphi^{P_1}(\vec{\psi}(|X|, i)) \\ &\leq 1 + \varphi_n(|X|) + \sum_{0 \leq i \leq \varphi_n(|X|)} \varphi^{P_1}(\vec{\psi}(|X|, i)) \\ &\quad \text{(d'après le lemme 3.3.4)} \end{aligned}$$

Il faut donc prendre :

$$\varphi^P(\vec{x}) = 1 + \varphi_n(\vec{x}) + \sum_{0 \leq i \leq \varphi_n(\vec{x})} \varphi^{P_1}(\vec{\psi}(\vec{x}, i)) \in \mathcal{PR}$$

car $\varphi_n \in \mathcal{PR}$ d'après le lemme 3.3.4, et $\varphi^{P_1} \in \mathcal{PR}$ d'après l'hypothèse d'induction.

□

3.4 Temps polynomial

Nous avons mis la définition de la classe en espace dans la section dédiée au temps primitif récursif car nous n'en aurons plus besoin par la suite, en particulier pour le temps polynomial.

En effet, comme annoncé, nous n'aurons besoin que d'imposer que les bornes n des commandes `loop` n soient des symboles statiques, c'est-à-dire ne pouvant être mis à jour.

Ainsi, il n'est pas nécessaire d'ajouter de contrainte sur l'espace, puisqu'il ne sera plus possible de transformer une augmentation d'espace en une augmentation de temps.

Bornes statiques

Exemple 3.4.1. (Un programme en temps exponentiel)

Le programme de la table 3.5 p.80 est bien dans `Loop` (voir p.43).

Chaque $x := r;$ `loop` x $\{r := r + 1;\}$ transforme r en $2r$, donc en commençant à $r = 1$ et en itérant n fois le programme calcule $r = 2^n$.

Or nous avons remarqué p.43 que la valeur d'une variable d'un programme de `Loop` est toujours bornée par la somme des valeurs initiales avec le nombre d'étapes déjà faites.

Ainsi, il faut bien un temps exponentiel pour construire la valeur $r = 2^n$.

TABLE 3.5 – Un programme de `Loop` pour l'exponentielle

$$
\begin{array}{l}
r := 1 \\
\texttt{loop}\ n \\
\quad x := r \\
\quad \texttt{loop}\ x \\
\quad\quad r := r + 1
\end{array}
$$

Dans son article [Nee03] Neergaard suggère que le temps exponentiel soit dû au fait que dans la boucle `loop` n la variable x puisse à la fois être mise à jour et serve à borner la boucle `loop` x.

Cette séparation entre les variables « sûres » pouvant être mises à jour et les variables « normales » pouvant contrôler les boucles n'est d'ailleurs pas sans rappeler la classe $\mathcal{B}$ de Bellantoni et Cook [BC92].

Neergaard a donc proposé son langage `PLoop` dont le temps d'exécution est polynomial. Il s'agit d'un langage `Loop` utilisant des listes comme structure de données et surtout imposant pour tous les programmes P la condition suivante :

« Pour chaque commande `loop` n P_1 de P : $Bound(P_1) \cap Upd(P_1) = \varnothing$ »

où $Upd(P_1)$ est l'ensemble des symboles mis à jour dans P_1 et où $Bound(P)$ est défini par induction :

Définition 3.4.2. (Variables bornant les boucles)

$$
\begin{aligned}
Bound(\{\}) &=_{def} \{\} \\
Bound(\{c; s\}) &=_{def} Bound(c) \cup Bound(\{s\})
\end{aligned}
$$

$$
\begin{aligned}
Bound(f t_1 \ldots t_k := t_0) &=_{def} \{\} \\
Bound(\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}) &=_{def} Bound(\{s_1\}) \cup Bound(\{s_2\}) \\
Bound(\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}) &=_{def} \{n\} \cup Bound(\{s_1\})
\end{aligned}
$$

Par simplicité nous avons posé à la p.46 que seules des variables pouvaient borner les boucles, donc $Bound(P)$ est un ensemble de variables.

Remarque. $Bound(P_1 P_2) = Bound(P_1) \cup Bound(P_2)$

Neergaard fait remarquer que sa contrainte peut être reformulée en remplaçant le « pour chaque commande `loop` n P_1 » par « les commandes `loop` n P_1 les plus extérieures ». Cependant, il reste possible de composer deux boucles, en utilisant comme borne pour la seconde un résultat calculé dans la première (voir la table 3.6 p.80).

TABLE 3.6 – Un programme pour l'exponentielle respectant la contrainte de Neergaard

$$
\begin{array}{l}
r := 1 \\
\texttt{loop}\ n \\
\quad r := 2 \times r \\
\texttt{loop}\ r
\end{array}
$$

Toutefois ce programme n'est pas un programme de `PLoop`, car « doubler » n'est pas une fonction de son langage. D'ailleurs ici nous ne l'utilisons directement que pour cacher sémantiquement la boucle qui a été nécessaire pour la calculer dans l'exemple précédent.

Toutefois, une telle fonction peut être considérée comme raisonnable, par exemple dans le modèle des entiers binaires (voir p.144) où elle ne correspond qu'à un décalage de bit [1].

En fait, dans `PLoop` l'espace ne peut grandir que polynomialement dans une boucle. Cela est dû au fait que pour tout terme $t \in T(P)$ nous avons $|\overline{t}^X| \leq max|X| + c_t$.

Rappel. Dans notre formalisme (voir p.142) si f est un constructeur alors :

$$|\overline{f}^X(\overline{t_1}^X, \ldots, \overline{t_\alpha}^X)| = 1 + \sum_{i=1}^{\alpha} |\overline{t_i}^X|$$

Ainsi, pour avoir le même genre de borne il faudrait se contenter des constructeurs unaires. Neergaard lui utilise des listes telles que :

$$|(d, e)| = max(|d|, |e|) \text{ et non } 1 + |d| + |e|$$

Toutefois, indépendamment des différences de formalisme pour les structures de données, le langage `PLoop` interdit des opérations élémentaires comme l'addition. Donc, en un sens, nous le voyons plutôt comme une sorte de langage d'assembleur, alors qu'une approche algorithmique nécessiterait davantage un langage de « haut niveau ».

Ainsi, au lieu de restreindre l'espace nous durcissons davantage la condition syntaxique de Neergaard sur les programmes impératifs : une variable de boucle ne sera pas seulement interdite de mise à jour dans une boucle, mais dans tout le programme.

Rappel. $Stat(P)$ est l'ensemble des symboles statiques (ne pouvant être mis à jour) du programme P.

Définition 3.4.3. (Restriction syntaxique de `LoopC`)

$$\texttt{LoopC}_{\texttt{stat}} =_{def} \{P \in \texttt{LoopC} \mid Bound(P) \subseteq Stat(P)\}$$

Ainsi, dans $\texttt{LoopC}_{\texttt{stat}}$ les bornes des boucles sont fixées à l'état initial. Certes, la discipline de programmation est élevée [2], mais contrairement au cas $\mathcal{PR}$ elle permet d'obtenir des temps polynomiaux sans faire aucune hypothèse sur les fonctions disponibles.

Ce qui est remarquable est qu'une contrainte syntaxique aussi simple soit suffisante non seulement pour garantir le temps polynomial quelles que soient les structures de données utilisées mais aussi pour simuler tous les algorithmes (voir chapitre suivant).

Nous irons même plus loin en montrant que le degré du polynôme correspond à la profondeur du programme :

Définition 3.4.4. (Profondeur d'un programme de `LoopC`)

$$\begin{aligned}
depth(\{\}) &=_{def} 0 \\
depth(\{c; s\}) &=_{def} max(depth(c), depth(\{s\}))
\end{aligned}$$

$$\begin{aligned}
depth(ft_1 \ldots t_k := t_0) &=_{def} 0 \\
depth(\texttt{if } F\ \{s_1\} \texttt{ else } \{s_2\}) &=_{def} max(depth(\{s_1\}), depth(\{s_2\})) \\
depth(\texttt{loop } n \texttt{ except } F\ \{s_1\}) &=_{def} 1 + depth(\{s_1\}) \\
&\qquad \text{si } n \in Dyn(P) \sqcup Init(P) \\
&=_{def} depth(\{s_1\}) \\
&\qquad \text{si } n \in Cons(P) \sqcup Oper(P)
\end{aligned}$$

1. ↑ Mais dans ce cas on pourrait s'attendre à ce qu'une boucle sur un entier binaire sorte les bits un par un donc fasse un nombre d'itérations correspondant à la longueur et non à la valeur de l'entier.
2. ↑ Cela revient à ne pouvoir utiliser des boucles que pour parcourir les entrées.

Remarque. $depth(P_1P_2) = max(depth(P_1), depth(P_2))$

Cette définition de la profondeur peut sembler étrange, car nous la distinguons de l'imbrication (voir la définition 4.2.3 p.95) notamment en ne comptant pas les boucles ayant une borne uniforme, que nous interprétons comme de la simple duplication de code :

Exemple 3.4.5. Les programmes de la table 3.7 p.82 ont une imbrication différente, mais la même profondeur.

TABLE 3.7 – Différence entre imbrication et profondeur

$r := 0$	$r := 0$
`loop` 3	$r := r + 1$
$\quad r := r + 1$	$r := r + 1$
	$r := r + 1$

Un fragment polynomial

Chez Neegaard et Bellantoni-Cook les variables sont distinguées entre :

1. Les variables « sûres » pouvant être mises à jour.
2. Les variables « normales » pouvant servir de borne.

Dans notre formalisme il s'agit de la séparation $\mathcal{L}(P) = Dyn(P) \sqcup Stat(P)$. Cependant, tous les symboles statiques ne contribuent pas à la taille de l'état :

Rappel. de la définition p.32 :

$$|X| = (|f|_X)_{f \in Dyn(A) \sqcup Init(A)}, \text{ où } |f|_X = \sup_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$$

Ainsi, les variables « normales » correspondent en fait à l'ensemble $Init(P)$.

Remarque. Si P_1 est un sous-programme de P alors $Bound(P_1) \subseteq Bound(P)$ et $Stat(P_1) \subseteq Stat(P)$. En particulier, si $P \in \texttt{LoopC}_{\texttt{stat}}$ alors $P_1 \in \texttt{LoopC}_{\texttt{stat}}$.

Dans la proposition suivante, nous utiliserons cette remarque pour noter tout simplement $Init$ l'ensemble de variables $Init(P)$ utilisé pour les bornes. Ainsi, nous pourrons faire la preuve par induction en utilisant le fait que si P_1 est un sous-programme de P alors $Init(P_1) \subseteq Init$:

Notation. Soit $|X|_{Init}$ le k-uplet $(|f|_X)_{f \in Init}$, c'est-à-dire $|X|$ tronqué aux seuls symboles de $Init$.

Proposition 3.4.6. *(Temps polynomial)*
Pour tout $P \in \texttt{LoopC}_{\texttt{stat}}$, il existe $\varphi_P \in \mathcal{P}ol$ telle que pour tout X :

1. $|P(X)|_{Init} = |X|_{Init}$
2. $time(P, X) \leq \varphi_P(|X|_{Init})$
3. $deg(\varphi_P) = depth(P)$

Démonstration. Par induction sur P :

$\boldsymbol{P = \{\}}$

$\varphi_P = 0$ convient :

1. $\{\}(X) = X$ donc $|P(X)|_{Init} = |X|_{Init}$
2. $time(\{\}, X) = 0$ donc $time(P, X) \leq 0$
3. $deg(0) = 0 = depth(\{\})$

$\boldsymbol{P = P_1 P_2}$

Supposons par induction que la proposition soit vraie pour P_1 et P_2.

Montrons-la pour $P = P_1 P_2$ en prenant $\varphi_{P_1 P_2} = \varphi_{P_1} + \varphi_{P_2}$.

D'après la proposition 2.2.9 p.49 :

1. $P_1 P_2(X) = P_2(P_1(X))$
   Donc $|P_1 P_2(X)|_{Init} = |P_2(P_1(X))|_{Init} = |P_1(X)|_{Init} = |X|_{Init}$
2. $time(P_1 P_2, X) = time(P_1, X) + time(P_2, P_1(X))$, donc :
$$\begin{aligned} time(P_1 P_2, X) &\leq \varphi_{P_1}(|X|_{Init}) + \varphi_{P_2}(|P_1(X)|_{Init}) \\ &= \varphi_{P_1}(|X|_{Init}) + \varphi_{P_2}(|X|_{Init}) \end{aligned}$$
3. $$\begin{aligned} deg(\varphi_{P_1 P_2}) &= deg(\varphi_{P_1} + \varphi_{P_2}) \\ &= max(deg(\varphi_{P_1}), deg(\varphi_{P_2})) \\ &= max(depth(P_1), depth(P_2)) \\ &= depth(P_1 P_2) \end{aligned}$$

Il ne reste alors plus qu'à prouver la proposition pour les commandes seules en utilisant l'hypothèse d'induction :

$\boldsymbol{P = f t_1 \ldots t_k := t_0}$

$\varphi_P = 1$ convient :

1. Le seul changement est que $\overline{f}^{f t_1 \ldots t_k := t_0(X)}(\overline{t_1}^X, \ldots, \overline{t_k}^X) = \overline{t_0}^X$
   Or f est dynamique donc n'est pas dans $Init$.
   D'où $|P(X)|_{Init} = |X|_{Init}$
2. $time(f t_1 \ldots t_k := t_0, X) = 1$ donc $time(P, X) \leq 1$
3. $deg(1) = 0 = depth(f t_1 \ldots t_k := t_0)$

$\boldsymbol{P =}$ `if` $\boldsymbol{F}$ `then` $\boldsymbol{P_1}$ `else` $\boldsymbol{P_2}$

Montrons que $\varphi_P = 1 + \varphi_{P_1} + \varphi_{P_2}$ convient :

$P \star X \succ P_i \star X$,

où $i = 1$ si F est vraie dans X et $i = 2$ si F est fausse dans X

1. $P(X) = P_i(X)$
   Or $|P_1(X)|_{Init} = |X|_{Init}$ et $|P_2(X)|_{Init} = |X|_{Init}$
   Donc $|P(X)|_{Init} = |X|_{Init}$
2. $time(P, X) = 1 + time(P_i, X) \leq 1 + time(P_1, X) + time(P_2, X)$
   Or $time(P_1, X) \leq \varphi_{P_1}(|X|_{Init})$ et $time(P_2, X) \leq \varphi_{P_2}(|X|_{Init})$
   Donc $time(P, X) \leq 1 + \varphi_{P_1}(|X|_{Init}) + \varphi_{P_2}(|X|_{Init})$
3. $$\begin{aligned} deg(\varphi_P) &= deg(1 + \varphi_{P_1} + \varphi_{P_2}) \\ &= max(deg(\varphi_{P_1}), deg(\varphi_{P_2})) \\ &= max(depth(P_1), depth(P_2)) \\ &= depth(P) \end{aligned}$$

$\boldsymbol{P} = \texttt{loop}\ \boldsymbol{n}\ \texttt{except}\ \boldsymbol{F}\ \boldsymbol{P_1}$

En reprenant la preuve du corollaire 2.2.10 p.50 nous avons que :

1. $P(X) = P_1^{\overline{n}^X}(X) \oplus (i, 0)$
2. $time(P, X) = \sum_{0 \leq i \leq \overline{n}^X - 1}(1 + time(P_1, P_1^i(X))) + 1$

Donc :

1. $P(X) = P_1^{\overline{n}^X}(X) \oplus (i, 0)$
   i n'est pas statique donc $|P(X)|_{Init} = |P_1^{\overline{n}^X}(X)|_{Init}$
   Or $|P_1(X)|_{Init} = |X|_{Init}$
   Donc par récurrence sur $\overline{n}^X$: $|P_1^{\overline{n}^X}(X)|_{Init} = |X|_{Init}$
   D'où $|P(X)|_{Init} = |P_1^{\overline{n}^X}(X)|_{Init} = |X|_{Init}$
2. $time(P, X) = 1 + \overline{n}^X + \sum_{0 \leq i \leq \overline{n}^X - 1} time(P_1, P_1^i(X))$
   Or $time(P_1, X) \leq \varphi_{P_1}(|X|_{Init})$
   D'où $time(P_1, P_1^i(X)) \leq \varphi_{P_1}(|P_1^i(X)|_{Init}) = \varphi_{P_1}(|X|_{Init})$, et :

$$\begin{aligned} time(P, X) &\leq 1 + \overline{n}^X + \textstyle\sum_{0 \leq i \leq \overline{n}^X - 1} \varphi_{P_1}(|X|_{Init}) \\ &= 1 + \overline{n}^X \times (1 + \varphi_{P_1}(|X|_{Init}))\ (1) \\ &\leq 1 + (\overline{n}^X + 1) \times (1 + \varphi_{P_1}(|X|_{Init}))\ (2) \end{aligned}$$

   Ces deux inégalité correspondent aux deux cas suivants :

   $\boldsymbol{n \in Cons(P) \sqcup Oper(P)}$
   Alors il existe une constante $c_n \in \mathbb{N}_1$ telle que $\overline{n}^X = c_n$
   D'où (1) : $\varphi_P(\vec{x}) = 1 + c_n \times (1 + \varphi_{P_1}(\vec{x}))$ convient.

   $\boldsymbol{n \in Init(P)}$
   Comme $Init(P) \subseteq Stat$, et que $|\overline{n}^X| = \overline{n}^X + 1$, nous avons $\overline{n}^X + 1 \in |X|_{Init}$ (disons le j-ième élément des k éléments)
   D'où (2) : $\varphi_P(\vec{x}) = 1 + x_j \times (1 + \varphi_{P_1}(\vec{x}))$ convient.
3. Le degré dépend des cas :

   $\boldsymbol{n \in Cons(P) \sqcup Oper(P)}$

$$\begin{aligned} deg(\varphi_P) &= deg(1 + c_n \times (1 + \varphi_{P_1})) \\ &= deg(\varphi_{P_1}) \\ &= depth(P_1) \\ &= depth(P) \end{aligned}$$

   $\boldsymbol{n \in Init(P)}$

$$\begin{aligned} deg(\varphi_P) &= deg(1 + \pi_j^k \times (1 + \varphi_{P_1})) \\ &= 1 + deg(\varphi_{P_1}) \\ &= 1 + depth(P_1) \\ &= depth(P) \end{aligned}$$

□

Ainsi, comme $time(P, X) \leq \varphi_P(|X|_{Init(P)})$, en utilisant les projections nous pouvons écrire $time(P, X) \leq \tilde{\varphi}_P(|X|)$, où $\tilde{\varphi}_P$ est polynomiale de degré $depth(P)$.

Remarque. Les fonctions linéaires ne sont qu'un cas particulier des fonctions polynomiales pour $depth(P) \leq 1$, c'est-à-dire en interdisant l'imbrication de « vraies » boucles (celles contribuant à la profondeur définie p.81).

Chapitre 4

Simulation des ASMs

Sommaire

4.1 Le programme noyau **85**
Traduction syntaxique 86
Le programme simulant un pas 87
4.2 Le programme coquille **92**
Simulation des complexités en temps 92
Condition d'arrêt 93
4.3 Le programme de la simulation **97**
Insertion de programme 97
La simulation 100

Le but de ce chapitre est de montrer que les programmes impératifs peuvent simuler les ASMs au sens de la définition 2.3.5 p.56. Pour cela nous devons procéder par étapes :

1. Un programme d'ASM Π décrit comment faire une étape de l'algorithme. Ainsi, dans la première section nous présenterons une traduction fidèle de Π en temps que programme impératif capable de simuler une étape de l'ASM : c'est le programme noyau[1] P_Π.
2. Pour simuler fidèlement toute une exécution de l'ASM, il faut répéter le programme noyau suffisamment de fois. Pour cela, dans la seconde section, nous utiliserons la complexité c_Π de l'algorithme afin de construire un programme durant assez longtemps : c'est le programme coquille P_{c_Π}.
3. Nous expliquerons ensuite comment « mélanger » le programme noyau P_Π et le programme coquille P_{c_Π} pour obtenir le programme final à la troisième section, et prouverons que les conditions de notre simulation raisonnable sont vérifiées.

4.1 Le programme noyau

Soit Π le programme d'ASM à simuler.

Le but de cette section est de construire le programme noyau P_Π dont l'exécution pourra simuler correctement une étape de Π.

1. ↑ Les appellations « coquille » et « noyau » sont issues de [APV10].

Traduction syntaxique

Dans la mesure où `ASM` et `Imp` ont les mêmes commandes pour les mises à jour et les conditionnelles, il est tentant de simplement traduire les programmes d'ASM de cette façon :

Définition 4.1.1. (Traduction commande par commande des ASMs)

$$\begin{aligned}(ft_1 \dots t_k := t_0)^{tr} &=_{def} \{ft_1 \dots t_k := t_0;\} \\ (\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})^{tr} &=_{def} \{\texttt{if } F \texttt{ then } \Pi_1^{tr} \texttt{ else } \Pi_2^{tr};\} \\ (\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar})^{tr} &=_{def} \Pi_1^{tr} \ \dots \ \Pi_n^{tr}\end{aligned}$$

La traduction des commandes $\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar}$ est une composition (voir p.48) de programmes $\Pi_1^{tr} \ \dots \ \Pi_n^{tr}$. Le problème est que pour les ASMs les commandes sont effectuées simultanément, alors que pour les programmes impératifs elles sont effectuées séquentiellement. Ainsi la traduction commande par commande n'est pas fidèle :

Exemple 4.1.2. (Traduction non fidèle)
Soit le programme d'ASM suivant :

$$\Pi = \texttt{par } x := y \| y := x \texttt{ endpar}$$

Nous avons que pour tout état X :

$$\tau_\Pi(X) = X \oplus \Delta(\Pi, X) = X \oplus \{(x, \overline{y}^X), (y, \overline{x}^X)\}$$

Or la traduction commande par commande de Π est :

$$\Pi^{tr} = \{x := y; y := x;\}$$

Donc une exécution de Π^{tr} serait :

$$\begin{aligned} & \{x := y; y := x;\} \star X \\ \succ \quad & \{y := x;\} \star X \oplus \{(x, \overline{y}^X)\} \\ \succ \quad & \{\} \star X \oplus \{(x, \overline{y}^X), (y, \overline{y}^X)\}\end{aligned}$$

Or la mise à jour $(y, \overline{y}^X)$ est triviale sur X, d'où $\Pi^{tr}(X) = X \oplus (x, \overline{y}^X)$. Donc dans le cas où $\overline{x}^X \neq \overline{y}^X$ nous avons :

$$\tau_\Pi(X) = X \oplus \{(x, \overline{y}^X), (y, \overline{x}^X)\} \neq X \oplus (x, \overline{y}^X) = \Pi^{tr}(X)$$

Pour capturer la simultanéité des programmes d'ASM, nous devons stocker dans des variables temporaires la valeur des termes lus. Plus précisément, pour chaque terme $t \in Read(\Pi)$ (voir p.37) nous devons introduire une variable temporaire v_t et la substituer dans tout le programme là où t est lu :

Définition 4.1.3. (Substitution d'un terme par une variable)

$$\begin{aligned}(ft_1 \dots t_k := t_0)[v/t] &=_{def} ft_1[v/t] \dots t_k[v/t] := t_0[v/t] \\ (\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})[v/t] &=_{def} \texttt{if } F[v/t] \texttt{ then } \Pi_1[v/t] \\ &\qquad \texttt{else } \Pi_2[v/t] \texttt{ endif} \\ (\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar})[v/t] &=_{def} \texttt{par } \Pi_1[v/t] \| \dots \| \Pi_n[v/t] \texttt{ endpar}\end{aligned}$$

$$\text{Où } t_1[v/t_2] =_{def} \begin{cases} v & \text{si } t_1 = t_2 \\ t_1 & \text{sinon} \end{cases}$$

Remarque. Comme les variables temporaires sont fraîches, si t_1 et t_2 sont des termes distincts alors $\Pi[v_{t_1}/t_1][v_{t_2}/t_2] = \Pi[v_{t_2}/t_2][v_{t_1}/t_1]$.

En particulier pour l'ensemble des termes $\vec{t}$ de $Read(\Pi)$, la notation $\Pi[\vec{v_t}/\vec{t}]$ ne sera pas ambigüe.

Exemple 4.1.4. (Traduction fidèle)

Reprenons notre exemple précédent.

Soient v_x et v_y deux nouvelles variables, respectivement initialisées à $\overline{x}^X$ et $\overline{y}^X$. Le programme obtenu en substituant les variables lues (mais non écrites !) x et y par les variables fraîches v_x et v_y correspondantes est donc :

$$\Pi^{tr}[v_x/x, v_y/y] = \{x := v_y; y := v_x;\}$$

Et l'exécution de ce programme donne bien :

$$\begin{array}{ll} & \{x := v_y; y := v_x;\} \star X \\ \succ & \qquad \{y := v_x;\} \star X \oplus \{(x, \overline{y}^X)\} \\ \succ & \qquad\qquad \{\} \star X \oplus \{(x, \overline{y}^X), (y, \overline{x}^X)\} \end{array}$$

D'où pour tout état X :

$$\tau_\Pi(X) = X \oplus \{(x, \overline{y}^X), (y, \overline{x}^X)\} = \Pi[v_x/x, v_y/y]^{tr}(X)$$

D'après la remarque p.42 pour tout programme d'ASM il existe un programme d'ASM équivalent et sous forme normale. Ainsi, nous pouvons supposer ici que le programme Π est déjà sous forme normale.

En utilisant la substitution 4.1.3 des termes de $Read(\Pi)$ puis la traduction 4.1.1 des commandes de Π nous obtenons la traduction $\Pi[\vec{v_t}/\vec{t}]^{tr}$ détaillée à la table 4.1 p.88.

Le programme simulant un pas

Il reste cependant deux problèmes :

1. Les variables $v_{t_1}, \ldots, v_{t_r}$ ne sont pas encore initialisées aux valeurs de $\{t_1, \ldots, t_r\} = Read(\Pi)$.
2. Selon la définition 2.3.5 p.56 la dilatation temporelle devrait être constante ce qui nécessiterait que le P_Π désiré fasse le même nombre d'étapes quel que soit l'état considéré.

Le premier problème se pose réellement car comme suggéré à la p.54 les variables temporaires ne devraient pas être initialisées avec des informations portant sur le calcul en cours, typiquement le résultat.

En fait nous nous sommes restreints ou bien à des initialisations uniformes (par exemple pour la traduction p.65 b_P est vraie et les autres b_{P_j} sont faux dans l'état initial) ou bien à la taille des entrées (cas que nous aborderons à la section suivante).

Ainsi, pour régler le premier problème, nous ne donnerons pas de valeur particulière aux v_t à l'initialisation mais nous ferons une initialisation explicite en ajoutant le programme $\vec{v_t} := \vec{t}$ avant le programme $\Pi[\vec{v_t}/\vec{t}]^{tr}$.

Remarque. Dans le programme P_Π désiré les variables $\vec{v_t}$ ne sont mises à jour que durant l'initialisation, donc elles contiennent la valeur initiale des $\vec{t}$:

$$\overline{v_t}^{P_\Pi(X)} = \overline{t}^X$$

TABLE 4.1 – Traduction de Π

$\Pi =$	$\Pi[\vec{v}_t/\vec{t}]^{tr} = \{$
`if` F_1 `then`	`if` v_{F_1} `then`
`par`	$\{$
$f_1^1(\vec{t}_1^1) := t_1^1$	$f_1^1(\vec{v}_{t_1^1}) := v_{t_1^1}$;
$f_2^1(\vec{t}_2^1) := t_2^1$	$f_2^1(\vec{v}_{t_2^1}) := v_{t_2^1}$;
$\vdots$	$\vdots$
$f_{m_1}^1(\vec{t}_{m_1}^1) := t_{m_1}^1$	$f_{m_1}^1(\vec{v}_{t_{m_1}^1}) := v_{t_{m_1}^1}$;
`endpar`	$\}$
`else`	`else` $\{$
`if` F_2 `then`	`if` v_{F_2} `then`
`par`	$\{$
$f_1^2(\vec{t}_1^2) := t_1^2$	$f_1^2(\vec{v}_{t_1^2}) := v_{t_1^2}$;
$f_2^2(\vec{t}_2^2) := t_2^2$	$f_2^2(\vec{v}_{t_2^2}) := v_{t_2^2}$;
$\vdots$	$\vdots$
$f_{m_2}^2(\vec{t}_{m_2}^2) := t_{m_2}^2$	$f_{m_2}^2(\vec{v}_{t_{m_2}^2}) := v_{t_{m_2}^2}$;
`endpar`	$\}$
$\vdots$	$\vdots$
`else`	`else` $\{$
`if` F_c `then`	`if` v_{F_c} `then`
`par`	$\{$
$f_1^c(\vec{t}_1^c) := t_1^c$	$f_1^c(\vec{v}_{t_1^c}) := v_{t_1^c}$;
$f_2^c(\vec{t}_2^c) := t_2^c$	$f_2^c(\vec{v}_{t_2^c}) := v_{t_2^c}$;
$\vdots$	$\vdots$
$f_{m_c}^c(\vec{t}_{m_c}^c) := t_{m_c}^c$	$f_{m_c}^c(\vec{v}_{t_{m_c}^c}) := v_{t_{m_c}^c}$;
`endpar`	$\}$
`endif` ... `endif`	; $\}$ ... ; $\}$

En particulier, comme $P_\Pi^{i+1}(X) = P_\Pi(P_\Pi^i(X))$, si P_Π est répété nous avons qu'à l'étape $i+1$ ces variables contiennent la valeur des $\vec{t}$ à l'étape précédente i. Plus formellement, nous avons que pour tout $i \in \mathbb{N}$ et pour tout $t \in Read(\Pi)$:

$$\overline{v_t}^{P_\Pi^{i+1}(X)} = \overline{t}^{P_\Pi^i(X)}$$

Pour régler le second problème il nous faut ajouter des commandes ne faisant rien mais consommant du temps afin de « matelasser » [1] la fin de chaque bloc.

Notation.

$$\begin{aligned} \texttt{skip}\ 0 &=_{def} \epsilon \\ \texttt{skip}\ n+1 &=_{def} \texttt{if}\ \mathit{true}\ \{\};\texttt{skip}\ n \end{aligned}$$

`skip` n fait bien exactement n étapes sans changer l'état en cours, mais il reste à déterminer pour chaque bloc la valeur de ce n.

Les formules F_i sont des « gardes » (voir p.40), c'est-à-dire que pour tout état X une et une seule F_i est vraie. Dans ce cas exactement i étapes seront faites pour explorer les conditionnelles jusqu'à entrer dans le bloc de F_i. Puis exactement m_i mises à jour seront faites dans ce bloc, avant que le programme ne s'arrête.

Donc le nombre d'étapes faites est $i + m_i$.

Soit $m = max\{m_i \mid 1 \leq i \leq c\}$, qui d'une certaine façon mesure le **degré de parallélisme** de Π [2]. Ainsi, nous avons que le nombre d'étapes faites par $\Pi[\vec{v}_t/\vec{t}]^{tr}$ est forcément $\leq c + m$ [3].

Donc en ajoutant à la fin de chaque bloc la séquence `skip` n avec $n = (c+m)-(i+m_i)$, le nombre d'étapes faites sera de exactement de $c + m$ dans tous les cas possibles.

En comptant les r étapes nécessaires à l'initialisation $\vec{v}_t := \vec{t}$ des variables fraîches nous obtenons alors un temps uniforme $t_\Pi = r + c + m$ pour le programme P_Π (voir table 4.2 p.90), qui est la traduction d'un pas de l'ASM Π.

Soit X un état de l'ASM Π, enrichi avec les variables temporaires. Comme attendu le programme P_Π permet de simuler une étape de Π en un nombre constant d'étapes :

Proposition 4.1.5. *(Traduction d'un pas d'ASM)*

1. $(P_\Pi(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$
2. $time(P_\Pi, X) = r + c + m = t_\Pi$

où :

- *r est le nombre de termes lus par Π (voir p.37)*
- *c est le nombre d'états reconnus par Π (voir p.40)*
- *m est le degré de parallélisme de Π (voir p.89)*

Démonstration. Nous ne donnons ici que l'idée de la preuve, la preuve complète étant faite p.132.

L'initialisation requiert r étapes.

Pour chaque variable v_t et pour tout état Y après l'initialisation nous avons :

$$\overline{v_t}^Y = \overline{t}^X$$

En particulier pour chaque conditionnelle F_i : $\overline{v_{F_i}}^Y = \overline{F_i}^X$

1. ↑ Comme dans l'article [FZG10] de Marie Ferbus-Zanda et Serge Grigorieff.
2. ↑ On peut assimiler cela au nombre maximum de tâches appelées par le programme.
3. ↑ Cette borne n'est pas optimale car nous aurions pu borner par $max\{i+m_i \mid 1 \leq i \leq c\}$. Toutefois je trouve la borne $c + m$ plus parlante pour comprendre la complexité de P_Π.

TABLE 4.2 – Simulation d'un pas de Π

$P_\Pi = \{$
$\quad v_{t_1} := t_1$;
$\quad v_{t_2} := t_2$;
$\quad \vdots$
$\quad v_{t_r} := t_r$;
$\quad$ `if` v_{F_1} `then` $\{$
$\quad\quad f_1^1(\vec{v}_{t_1^1}) := v_{t_1^1}$;
$\quad\quad f_2^1(\vec{v}_{t_2^1}) := v_{t_2^1}$;
$\quad\quad \vdots$
$\quad\quad f_{m_1}^1(\vec{v}_{t_{m_1}^1}) := v_{t_{m_1}^1}$;
$\quad\quad$ `skip` $(c+m)-(1+m_1)$
$\quad \}$ `else` $\{$
$\quad$ `if` v_{F_2} `then` $\{$
$\quad\quad f_1^2(\vec{v}_{t_1^2}) := v_{t_1^2}$;
$\quad\quad f_2^2(\vec{v}_{t_2^2}) := v_{t_2^2}$;
$\quad\quad \vdots$
$\quad\quad f_{m_2}^2(\vec{v}_{t_{m_2}^2}) := v_{t_{m_2}^2}$;
$\quad\quad$ `skip` $(c+m)-(2+m_2)$
$\quad \}$ `else` $\{$
$\quad \vdots$
$\quad$ `if` v_{F_c} `then` $\{$
$\quad\quad f_1^c(\vec{v}_{t_1^c}) := v_{t_1^c}$;
$\quad\quad f_2^c(\vec{v}_{t_2^c}) := v_{t_2^c}$;
$\quad\quad \vdots$
$\quad\quad f_{m_c}^c(\vec{v}_{t_{m_c}^c}) := v_{t_{m_c}^c}$;
$\quad\quad$ `skip` $(c+m)-(c+m_c)$
$\quad \}$; $\ldots \}$;
$\}$

Comme les F_i sont des gardes, nous avons qu'une et une seule est vraie dans X, soit F_j cette formule.

Il faut exactement j étapes pour arriver au bloc de F_j, puis m_j étapes pour faire les mises à jour. Puis, à cause de la séquence `skip` $(c+m)-(j+m_j)$ nous avons que le temps d'exécution ne dépend que de Π :

$$time(P_\Pi, X) = r + (j + m_j) + (c + m) - (j + m_j) = r + c + m$$

Après l'initialisation des variables, les mises à jour faites dans le bloc sont bien $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$, donc :

$$\Delta(P_\Pi, X) = \{(v_t, \bar{t}^X) \mid t \in Read(\Pi)\} \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$$

Les variables temporaires ne sont mises à jour qu'une fois durant l'initialisation puis, comme l'ASM est sous forme normale, $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$ est consistant.

Donc P_Π est sans écrasement, et comme il est terminal nous avons d'après la proposition 2.2.12 :

$$\Delta(P_\Pi, X) = P_\Pi(X) \ominus X$$

Nous déduisons de ces deux égalités que :

$$(P_\Pi(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$$

□

Ainsi, il nous faut un nombre d'étapes ne dépendant que de Π pour simuler une étape de Π. En répétant le programme P_Π nous pouvons donc simuler l'exécution de Π :

Corollaire 4.1.6. $P^i_\Pi(X)|_{\mathcal{L}(\Pi)} = \tau^i_\Pi(X|_{\mathcal{L}(\Pi)})$

Démonstration. Par induction sur i :

$\boldsymbol{i = 0}$

$$P^0_\Pi(X)|_{\mathcal{L}(\Pi)} = X|_{\mathcal{L}(\Pi)} = \tau^0_\Pi(X|_{\mathcal{L}(\Pi)})$$

$\boldsymbol{i \Rightarrow i+1}$

$$\begin{aligned}
P^{i+1}_\Pi(X)|_{\mathcal{L}(\Pi)} &= P_\Pi(P^i_\Pi(X))|_{\mathcal{L}(\Pi)} \\
&= (P^i_\Pi(X) \oplus (P_\Pi(P^i_\Pi(X)) \ominus P^i_\Pi(X)))|_{\mathcal{L}(\Pi)} \\
&= P^i_\Pi(X)|_{\mathcal{L}(\Pi)} \oplus (P_\Pi(P^i_\Pi(X)) \ominus P^i_\Pi(X))|_{\mathcal{L}(\Pi)} \\
&\quad \text{(d'après le lemme A.1.10 p.120)} \\
&= P^i_\Pi(X)|_{\mathcal{L}(\Pi)} \oplus \Delta(\Pi, P^i_\Pi(X)|_{\mathcal{L}(\Pi)}) \\
&\quad \text{(d'après la proposition 4.1.5)} \\
&= \tau^i_\Pi(X|_{\mathcal{L}(\Pi)}) \oplus \Delta(\Pi, \tau^i_\Pi(X|_{\mathcal{L}(\Pi)})) \\
&\quad \text{(d'après l'hypothèse d'induction)} \\
&= \tau^{i+1}_\Pi(X|_{\mathcal{L}(\Pi)})
\end{aligned}$$

□

4.2 Le programme coquille

Maintenant que nous avons obtenu le programme noyau P_Π il faut le répéter assez longtemps pour simuler l'exécution de Π.

Soit c_Π la complexité (voir la définition 1.3.6 p.33) de l'ASM Π. Nous nous intéresserons aux cas où c_Π est quelconque, primitive récursive ou polynomiale

Le but de cette section est de montrer qu'il existe un programme impératif P_{c_Π}, le programme coquille, tel que :

1. $time(P_{c_\Pi}, X) = time(\Pi, X|_{\mathcal{L}(\Pi)})$
2. P_{c_Π} appartienne à un fragment de `Imp` caractéristique de la classe de c_Π.

Nous expliquerons ensuite à la section suivante comment mélanger les programmes noyau et coquille afin d'obtenir le programme permettant de simuler Π.

Simulation des complexités en temps

Pour l'instant, nous allons nous contenter d'un programme impératif P_{c_Π} tel que $time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$, et nous le modifierons par la suite pour qu'il s'arrête au bon moment.

Afin de caractériser les classes de complexité, le programme obtenu appartiendra à différents fragments `Imp` selon la complexité de c_Π :

c_Π quelconque

Comme il suffit d'avoir $time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$, un programme ne terminant jamais convient :

$$P_{c_\Pi} = \{\texttt{while}\ true\ \{\};\} \in \texttt{While}$$

$c_\Pi \in \mathcal{PR}$

Selon Meyer et Ritchie [MR67] il existe un programme $P_{\texttt{loop}} \in \texttt{Loop}$ (voir p.43) calculant $c_\Pi(\vec{n})$ en fonction de $\vec{n} = |X|$.

Il suffit donc d'initialiser des variables $n_1, \ldots, n_k$ à la taille des des entrées, d'exécuter $P_{\texttt{loop}}$ en stockant le résultat dans une variable r, et d'utiliser r comme borne d'une boucle pour convertir la valeur en temps de calcul. Ainsi, le programme suivant convient :

$$P_{c_\Pi} = \{r := P_{\texttt{loop}}(n_1, \ldots, n_k); \texttt{loop}\ r\ \{\};\} \in \texttt{Loop}$$

$c_\Pi \in \mathcal{P}ol$

Dans ce cas, la complexité de Π peut s'écrire de la façon suivante :

$$c_\Pi(|X|) = \sum_{0 \leq d_1 + \cdots + d_k \leq deg(c_\Pi)} c_{d_1,\ldots,d_k} \prod_{n_j \in |X|} n_j^{d_j}$$

Soit $P_{d_1,\ldots,d_k}$ le programme de la table 4.3 p.93 constitué d'imbrications successives de boucles et où $n_{d_1,\ldots,d_k}$ est initialisée à la valeur $c_{d_1,\ldots,d_k}$, et chaque n_j à la valeur du j-ième élément de $|X|$.

Aucune mise à jour n'est faite, donc la valeur des bornes ne changent pas durant l'exécution. Nous avons[1] pour tout X :

$$time(P_{d_1,\ldots,d_k}, X) \geq c_{d_1,\ldots,d_k} \prod_{n_j \in |X|} n_j^{d_j}$$

1. ↑ Il n'y a pas égalité car en fait $time(P_{d_1,\ldots,d_k}, X) = (c_{d_1,\ldots,d_k} + 1) \prod_{n_j \in |X|} (n_j + 1)^{d_j}$, les $+1$ venant de l'étape supplémentaire pour effacer une boucle quand elle a terminé.

TABLE 4.3 – Programme pour la complexité polynomiale

$$
\begin{array}{l}
\texttt{loop}\ n_{d_1,\ldots,d_k} \\
\quad \left.\begin{array}{l}
\texttt{loop}\ n_1 \\
\quad \ddots \\
\qquad \texttt{loop}\ n_1
\end{array}\right\} d_1 \text{ fois} \\
\qquad\qquad \ddots \\
\qquad\qquad\quad \left.\begin{array}{l}
\texttt{loop}\ n_k \\
\quad \ddots \\
\qquad \texttt{loop}\ n_k
\end{array}\right\} d_k \text{ fois}
\end{array}
$$

Soit P_{c_Π} la composition des $P_{d_1,\ldots,d_k}$, pour $0 \leq d_1 + \cdots + d_k \leq deg(c_\Pi)$.
D'où, d'après la proposition 2.2.9, nous avons pour tout X que :

$$time(P_{c_\Pi}, X) \geq \sum_{0 \leq d_1 + \cdots + d_k \leq deg(c_\Pi)} c_{d_1,\ldots,d_k} \prod_{n_j \in |X|} n_j^{d_j} = c_\Pi(|X|)$$

Comme aucune mise à jour n'est faite, les nouveaux symboles peuvent être pris statiques, donc $P_{c_\Pi} \in \texttt{LoopC}_{\texttt{stat}}$.
Dans le programme $P_{d_1,\ldots,d_k}$ le symbole $n_{d_1,\ldots,d_k}$ a une interprétation ne dépendant que de c_Π, contrairement aux symboles n_j dont l'interprétation dépend de l'état initial. Donc d'après la définition p.81 nous avons que :

$$depth(P_{d_1,\ldots,d_k}) = d_1 + \cdots + d_k$$

Comme P_{c_Π} est la composition des $P_{d_1,\ldots,d_k}$, sa profondeur est la profondeur maximale des $P_{d_1,\ldots,d_k}$, donc :

$$\begin{aligned}
depth(P_{c_\Pi}) &= \max_{0 \leq d_1 + \cdots + d_k \leq deg(c_\Pi)} \{depth(P_{d_1,\ldots,d_k})\} \\
&= \max_{0 \leq d_1 + \cdots + d_k \leq deg(c_\Pi)} \{d_1 + \cdots + d_k\} = deg(c_\Pi)
\end{aligned}$$

En particulier une complexité linéaire sera implémentée par un programme de profondeur ≤ 1.

Remarque. Dans les trois cas, les symboles utilisés dans le programme P_{c_Π} sont indépendants de ceux utilisés dans P_Π.

Nous n'avons utilisé pour l'instant dans ces programmes que des `while`, des `loop` non conditionnés et des mises à jour (dans le programme de Meyer et Ritchie). Et effectivement, les `if` et les `loop` conditionnés ne vont apparaître qu'à l'ajout de la condition d'arrêt :

Condition d'arrêt

Nous disposons donc à présent d'un programme P_{c_Π} tel que pour tout X :

$$time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$$

Le problème est que nous avons uniquement une borne, et pas la complexité elle-même, ce qui signifie que la simulation ne s'arrêterait pas au bon moment. Formellement, cela

revient à violer la troisième condition de la définition 2.3.5 p.56. Mais plus important, cela exclurait de fait certains bons algorithmes, par exemple le *Min* p.24.

Il nous faut donc deux choses :

1. Pouvoir détecter quand l'ASM a terminé son exécution.
2. Pouvoir arrêter le programme coquille à ce moment.

Nous obtenons le premier point à l'aide de la μ-formule de Π, que nous appelons ainsi en raison de sa similitude avec l'opérateur de minimisation μ utilisé pour les fonctions récursives (voir [CL03b]).

Notation. La μ-formule de Π est définie par :

$$F_\Pi =_{def} \bigwedge_{t \in Read(\Pi)} v_t = t$$

Rappel. de la p.23 :

$$time(A, X_0) =_{def} \begin{cases} min\{i \in \mathbb{N} \mid \tau_A^i(X_0) = \tau_A^{i+1}(X_0)\} & \text{si } \vec{X} \text{ est terminale} \\ \infty & \text{sinon} \end{cases}$$

Lemme 4.2.1. *(μ-formule)*

$$time(\Pi, X|_{\mathcal{L}(\Pi)}) = min\{i \in \mathbb{N} \mid \overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true\}$$

Démonstration. Les deux côtés de :

$$\tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}) \Leftrightarrow \overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true$$

sont prouvés p.133, en utilisant le lemme 2.1.4 p.37 sur $Read(\Pi)$. □

Comme la formule F_Π permet de détecter quand l'ASM s'est arrêtée, pour obtenir le second point nous modifions le programme P_{c_Π} pour qu'il arrête la simulation quand F_Π devient vraie.

Nous utilisons ici le fait que les `loop` de P_{c_Π} ne soient pas conditionnés et que le programme ne contienne pas de conditionnelles pour simplifier un peu la traduction :

Définition 4.2.2. (Condition d'arrêt)
$P^{F_{end}}$, le programme P s'arrêtant quand F_{end} est vraie, est défini par induction sur P :

$$\begin{aligned} \{\}^{F_{end}} &=_{def} \{\} \\ \{c; s\}^{F_{end}} &=_{def} \{(c)^{F_{end}}; \texttt{if } \neg F_{end} \texttt{ then } \{s\}^{F_{end}};\} \end{aligned}$$

$$\begin{aligned} (ft_1 \ldots t_k := t_0)^{F_{end}} &=_{def} ft_1 \ldots t_k := t_0 \\ (\texttt{while } F\ \{s_1\})^{F_{end}} &=_{def} \texttt{while } (F \wedge \neg F_{end})\ \{\texttt{if } \neg F_{end} \texttt{ then } \{s_1\}^{F_{end}};\} \\ (\texttt{loop } n\ \{s_1\})^{F_{end}} &=_{def} \texttt{loop } n \texttt{ except } F_{end}\ \{\texttt{if } \neg F_{end} \texttt{ then } \{s_1\}^{F_{end}};\} \end{aligned}$$

Remarque. L'encapsulation `if` $\neg F_{end}$ `then` $\{s_1\}^{F_{end}}$ pour le corps des boucles n'est pas strictement nécessaire puisque le programme a déjà testé si F_{end} était vraie ou non. Nous l'avons rajoutée pour « perdre » du temps et ainsi avoir uniformément deux étapes pour simuler une étape du programme d'origine.

Remarque. $depth(P^{F_{end}}) = depth(P)$ d'après la définition p.81.

Selon la remarque p.93 les symboles utilisés dans le programme P_{c_Π} sont indépendants de ceux utilisés dans P_Π, donc la valeur de F_Π n'est pas influencée par l'exécution de P_{c_Π}. D'après la définition de la transformation $P^{F_{end}}$ nous avons :

1. Si F_{end} est vraie alors $P^{F_{end}} \succ_2 \{\}$
2. Si F_{end} est fausse alors :

$$\begin{array}{ll}
\{ft_1 \dots t_k := t_0; s\}^{F_{end}} \succ_2 \{s\}^{F_{end}} & \\
\{\texttt{while } F\ \{s_1\}; s\}^{F_{end}} \succ_2 \{s_1\}^{F_{end}}\ \{\texttt{while } F\ \{s_1\}; s\}^{F_{end}} & \text{si } F \text{ est vraie} \\
\{\texttt{while } F\ \{s_1\}; s\}^{F_{end}} \succ_2 \{s\}^{F_{end}} & \text{si } F \text{ est fausse} \\
\quad \{\texttt{loop } n\ \{s_1\}; s\}^{F_{end}} \succ_2 \{s_1\}^{F_{end}}\ \{\texttt{loop } n\ \{s_1\}; s\}^{F_{end}} & \text{si } i < n \\
\quad \{\texttt{loop } n\ \{s_1\}; s\}^{F_{end}} \succ_2 \{s\}^{F_{end}} & \text{si } i = n
\end{array}$$

Nous avons ainsi « presque » $P^{F_{end}} \star X \succ_2 \tau_X^1(P)^{F_{end}} \star \tau_P^1(X)$, à l'exception des boucles qui séparent les traductions de programme, notamment :

Remarque. Si F_{end} est fausse alors $time(P^{F_{end}}, X) = 2 \times time(P, X)$.

Plus précisément, nous aurons une alternance entre programmes de la forme :

1. $\texttt{if } \neg F_{end} \texttt{ then } \{s_{P_1}\}^{F_{end}} P_2^{F_{end}} \dots\ P_k^{F_{end}}$
2. $P_1^{F_{end}} \dots\ P_k^{F_{end}}$

Nous allons montrer dans le lemme suivant que le k en question ne dépend que de l'imbrication, qui est à distinguer de la profondeur définie p.81 :

Définition 4.2.3. (Imbrication d'un programme impératif)

$$\begin{aligned}
nest(\{\}) &=_{def} 0 \\
nest(\{c; s\}) &=_{def} max(nest(c), nest(\{s\}))
\end{aligned}$$

$$\begin{aligned}
nest(ft_1 \dots t_k := t_0) &=_{def} 0 \\
nest(\texttt{if } F\ \{s_1\} \texttt{ else } \{s_2\}) &=_{def} max(nest(\{s_1\}), nest(\{s_2\})) \\
nest(\texttt{while } F\ \{s_1\}) &=_{def} 1 + nest(\{s_1\}) \\
nest(\texttt{loop } n \texttt{ except } F\ \{s_1\}) &=_{def} 1 + nest(\{s_1\})
\end{aligned}$$

Remarque.

$$nest(P_1 P_2) = max(nest(P_1), nest(P_2))$$

Deux remarques avant de prouver le lemme en question :

1. D'après la définition de la transformation 4.2.2 nous avons par induction immédiate sur P que $nest(P^{F_{end}}) = nest(P)$.
2. Par induction sur k nous avons que $P_1^{F_{end}} \dots\ P_k^{F_{end}}$ s'écrit de façon unique, c'est-à-dire que si $P_1^{F_{end}} \dots\ P_k^{F_{end}} = Q_1^{F_{end}} \dots\ Q_\ell^{F_{end}}$ et que les programmes sont non vides alors[1] $k = \ell$ et pour tout i nous avons $P_i = Q_i$.

Enfin, pour simplifier la présentation nous inversons les indices dans le lemme :

Lemme 4.2.4. *Si F_{end} est fausse alors :*

$$\begin{array}{rl}
 & P_k^{F_{end}}\ P_{k-1}^{F_{end}} \dots\ P_1^{F_{end}} \\
\succ & \texttt{if } \neg F_{end} \texttt{ then } \{s_{Q_\ell}\}^{F_{end}}\ Q_{\ell-1}^{F_{end}} \dots\ Q_1^{F_{end}} \\
\succ & Q_\ell^{F_{end}}\ Q_{\ell-1}^{F_{end}} \dots\ Q_1^{F_{end}}
\end{array}$$

tel que $\max\limits_{1 \le i \le k}\{nest(P_i) + i\} \ge \max\limits_{1 \le j \le \ell}\{nest(Q_j) + j\}$

1. ↑ Dans le lemme, comme on a une transition on ne peut avoir $k = 0$, en revanche on peut avoir $\ell = 0$ si c'est la dernière transition, et dans ce cas $s_{Q_\ell} = \epsilon$. Il est à noter que si $\ell = 0$ alors le maximum de droite vaut 0

Démonstration. La preuve est faite p.134 par cas sur P_k. □

Rappel. Nous avons obtenu à partir de la complexité c_Π de l'ASM Π un programme P_{c_Π} tel que $time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$.

En utilisant la transformation 4.2.2 avec $F_{end} = F_\Pi$, la μ-formule de Π définie p.94, nous obtenons le programme coquille $P_{c_\Pi}^{F_\Pi}$.

Nous expliquerons à la section suivante comment mélanger le programme noyau P_Π et le programme coquille $P_{c_\Pi}^{F_\Pi}$ pour simuler l'exécution de Π.

En attendant, nous savons d'après le lemme 4.2.1 p.94 que F_Π sera fausse tant que Π n'aura pas terminé mais finira par devenir vraie si l'exécution est terminale.

Ainsi, tant que F_Π est fausse nous pouvons itérer le lemme 4.2.4 en partant du programme coquille $P_{c_\Pi}^{F_\Pi}$. Au bout des i étapes nécessaires pour que F_Π devienne vraie, deux cas sont possibles :

1. $P_{c_\Pi}^{F_\Pi} \star X \succ_i Q_\ell^{F_\Pi} \dots\ Q_1^{F_\Pi} \star X'$
2. $P_{c_\Pi}^{F_\Pi} \star X \succ_i \texttt{if}\ \neg F_\Pi\ \texttt{then}\ \{s_{Q_\ell}\}^{F_\Pi}\ Q_{\ell-1}^{F_\Pi} \dots\ Q_1^{F_\Pi} \star X'$

Ainsi, nous avons d'après le lemme 4.2.4 que :

$$nest(P_{c_\Pi}) + 1 \geq max_{1 \leq j \leq \ell}\{nest(Q_j) + j\}$$

Cela nous permet d'obtenir une borne pour ℓ ne dépendant que de Π :

$$\ell \leq nest(Q_\ell) + \ell \leq max_{1 \leq j \leq \ell}\{nest(Q_j) + j\} \leq nest(P_{c_\Pi}) + 1$$

Comme remarqué p.95, si F_Π est vraie alors $P^{F_\Pi} \succ_2 \{\}$, d'où selon les cas :

1.
$$\begin{array}{lll} & Q_\ell^{F_\Pi}\ Q_{\ell-1}^{F_\Pi} \dots\ Q_1^{F_\Pi} \\ \succ_2 & \quad\ \ Q_{\ell-1}^{F_\Pi} \dots\ Q_1^{F_\Pi} \\ \vdots & \\ \succ_2 & \qquad\qquad\qquad \{\} \end{array}$$

D'où $Q_\ell^{F_\Pi} \dots\ Q_1^{F_\Pi} \succ_{2\times\ell} \{\}$

2.
$$\begin{array}{lll} & \texttt{if}\ \neg F_\Pi\ \texttt{then}\ \{s_{Q_\ell}\}^{F_\Pi}\ Q_{\ell-1}^{F_\Pi}\ Q_{\ell-2}^{F_\Pi} \dots\ Q_1^{F_\Pi} \\ \succ & \qquad\qquad\qquad\qquad\ \ Q_{\ell-1}^{F_\Pi}\ Q_{\ell-2}^{F_\Pi} \dots\ Q_1^{F_\Pi} \\ \succ_2 & \qquad\qquad\qquad\qquad\qquad\ \ Q_{\ell-2}^{F_\Pi} \dots\ Q_1^{F_\Pi} \\ \vdots & \\ \succ_2 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\} \end{array}$$

D'où[1] $\texttt{if}\ \neg F_\Pi\ \texttt{then}\ \{s_{Q_\ell}\}^{F_\Pi}\ Q_{\ell-1}^{F_\Pi} \dots\ Q_1^{F_\Pi} \succ_{1+2\times(\ell-1)} \{\}$

Donc, selon les cas, quand F_Π devient vraie il reste 2ℓ ou $2\ell - 1$ étapes à effectuer pour $P_{c_\Pi}^{F_\Pi}$, avec $\ell \leq nest(P_{c_\Pi}) + 1$. D'où dans tous les cas :

Corollaire 4.2.5. *À partir du moment où F_Π devient vraie :*

$$time(\tau_X^i(P), \tau_P^i(X)) \leq 2 \times (nest(P_{c_\Pi}) + 1)$$

où $P = P_{c_\Pi}^{F_\Pi}$ est le programme coquille.

Nous verrons à la prochaine section que le programme noyau P_Π est répété avant chaque étape du programme coquille $P_{c_\Pi}^{F_\Pi}$.

Comme le i du corollaire est le nombre d'étapes pour que F_Π devienne vraie, d'après le lemme 4.2.1 p.94 nous aurons donc qu'en fait $i = time(\Pi, X|_{\mathcal{L}(\Pi)})$.

1. ↑ Si $\ell = 0$ alors $\tau_X^i(P) = \texttt{if}\ \neg F_\Pi\ \texttt{then}\ \{\}$ qui termine toujours en une étape, donc qui vérifie aussi la borne $time(\tau_X^i(P), \tau_P^i(X)) \leq 2 \times (nest(P_{c_\Pi}) + 1)$.

4.3 Le programme de la simulation

Rappel. À partir de l'ASM Π nous avons obtenu :

1. Un programme noyau P_Π qui d'après la proposition p.89 simule une étape de Π en un temps t_Π constant (ne dépendant que de Π)

   Donc il nous faut répéter P_Π exactement $time(\Pi, X|_{\mathcal{L}(\Pi)})$ fois pour simuler l'exécution de Π commençant à l'état initial $X|_{\mathcal{L}(\Pi)}$.

2. Un programme coquille $P = P_{c_\Pi}^{F_\Pi}$ tel que :

   a. D'après le lemme p.94 :

   $$min\{i \in \mathbb{N} \mid \overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true\} = time(\Pi, X|_{\mathcal{L}(\Pi)})$$

   Donc en partant de l'état $P_\Pi(X)$, F_Π ne devient vraie qu'au bout d'exactement $time(\Pi, X|_{\mathcal{L}(\Pi)})$ répétitions de P_Π.

   b. D'après la remarque p.95 :

   $$\text{Si } F_\Pi \text{ est fausse alors } time(P, X) = 2 \times time(P_{c_\Pi}, X)$$

   Comme $time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$ nous avons donc que :

   $$\text{Si } F_\Pi \text{ est fausse alors } time(P, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$$

   Donc le programme coquille P dure assez longtemps pour que F_Π puisse devenir vraie.

   c. D'après le corollaire 4.2.5, à partir du moment où F_Π devient vraie :

   $$time(\tau_X^i(P), \tau_P^i(X)) \leq 2 \times (nest(P_{c_\Pi}) + 1)$$

   Nous avons donc une borne pour la durée de la fin de l'exécution.

Explicitons à présent le mélange annoncé entre le programme noyau P_Π et le programme coquille $P_{c_\Pi}^{F_\Pi}$: nous voulons que P_Π soit exécuté à chaque étape de $P_{c_\Pi}^{F_\Pi}$.

Insertion de programme

Pour cela, nous allons l'insérer syntaxiquement entre chaque commande de $P_{c_\Pi}^{F_\Pi}$, pour obtenir le programme noté $P_{c_\Pi}^{F_\Pi}[P_\Pi]$. Il reste cependant à déterminer si l'insertion de P_Π doit être faite avant ou après les commandes de $P_{c_\Pi}^{F_\Pi}$.

L'ASM Π fait $time(\Pi, X|_{\mathcal{L}(\Pi)})$ étapes mais il faut en fait $time(\Pi, X|_{\mathcal{L}(\Pi)})+1$ répétitions de P_Π pour que F_Π mette fin à l'exécution.

C'est pour cela, ainsi que pour initialiser proprement les variables v_t qui seront testées dans F_Π, que nous commençons l'exécution par une occurrence de P_Π. Ainsi nous utiliserons en fait P_Π $P_{c_\Pi}^{F_\Pi}[P_\Pi]$.

Puis nous exécutons $time(\Pi, X|_{\mathcal{L}(\Pi)})$ fois dans l'ordre une commande de $P_{c_\Pi}^{F_\Pi}$ puis une exécution de $P_{c_\Pi}^{F_\Pi}$, et ensuite les commandes de $P_{c_\Pi}^{F_\Pi}$ peuvent mettre fin au programme. L'insertion doit donc se faire après, contrairement à [MV09] :

Définition 4.3.1. (Insertion de programmes)
$P[Q]$, le programme où Q est inséré après chaque commande de P, est défini par induction sur P :

$$\begin{aligned} \{\}[Q] &=_{def} \{\} \\ \{c; s\}[Q] &=_{def} (c)[Q]\ \{s\}[Q] \end{aligned}$$

$$\begin{aligned} (ft_1 \dots t_k := t_0)[Q] &=_{def} ft_1 \dots t_k := t_0\ Q \\ (\texttt{if}\ F\ \texttt{then}\ P_1\ \texttt{else}\ P_2)[Q] &=_{def} \texttt{if}\ F\ \texttt{then}\ (Q\ P_1[Q])\ \texttt{else}\ (Q\ P_2[Q]) \\ (\texttt{while}\ F\ P_1)[Q] &=_{def} \texttt{while}\ F\ (Q\ P_1[Q]) \\ (\texttt{loop}\ n\ \texttt{except}\ F\ P_1)[Q] &=_{def} \texttt{loop}\ n\ \texttt{except}\ F\ (Q\ P_1[Q]) \end{aligned}$$

Remarque. $P_1[Q]P_2[Q] = (P_1P_2)[Q]$

L'insertion de Q dans P se comporte comme attendu :

Lemme 4.3.2. *(Sémantique de l'insertion)*
Si $P \neq \{\}$ et Q est terminal alors

$$P[Q] \star X \succ_t \tau_X^1(P)[Q] \star Q(\tau_P^1(X))$$

avec $t = 1 + time(Q, \tau_P^1(X))$

Démonstration. La preuve est faite p.135 par induction sur P. □

Rappel. Selon la remarque p.93, les symboles de P_{c_Π} et P_Π sont disjoints.

En prenant $P = P_{c_\Pi}^{F_\Pi}$ et $Q = P_\Pi$ les deux programmes ne « communiquent » que par F_Π, donc par la suite nous noterons $Q(\tau_P^1(X))$ en la partageant en deux parties disjointes : $Q(\tau_P^1(X)) = \tau_P^1(X) \sqcup Q(X)$.

L'idée est donc d'itérer ce lemme pour avoir $\tau_P^i(X) \sqcup P_\Pi^i(X)$ jusqu'à ce que F_Π devienne fausse. Ainsi nous simulons l'exécution $P_\Pi^i(X)$ avec une dilatation temporelle de $d = 1 + t_\Pi$.

Il reste cependant un dernier problème à résoudre : la durée de la fin d'exécution. En effet, d'après le lemme 4.2.5 p.96, à partir du moment où F_Π devient vraie :

$$time(\tau_X^i(P), \tau_P^i(X)) \leq 2 \times (nest(P_{c_\Pi}) + 1)$$

Le problème est que nous pouvons borner le nombre d'étapes encore à faire, mais nous ne pouvons le connaître à l'avance puisqu'il dépend de l'état initial. Ainsi nous avons certes obtenu selon notre définition de la simulation p.56 un d uniforme, mais la constante de fin e elle ne l'est pas.

Pour la rendre uniforme nous allons compter dans une variable fraîche i_{end} initialisée à 0 le nombre d'étapes parcourues par le programme une fois que l'ASM est arrêtée, et ensuite utiliser la valeur de i_{end} pour « matelasser » la fin du programme.

Ainsi, le programme noyau n'est en fait pas P_Π, mais le programme :

$$Q = \texttt{if}\ \neg F_\Pi\ \texttt{then}\ P_\Pi\ \texttt{else}\ \{i_{end} := i_{end} + 3;\} \in \texttt{While} \cap \texttt{LoopC}$$

Le +3 vient du fait qu'en plus de l'étape de $P_{c_\Pi}^{F_\Pi}$ une deuxième étape est nécessaire pour entrer dans le `else` et une troisième pour mettre i_{end} à jour.

D'après le corollaire 2.2.10 p.50 Q est terminal et donc nous pouvons lui appliquer le lemme 4.3.2 dans les deux cas :

1. si F_Π est fausse alors $time(Q, Y) = 1 + t_\Pi$

2. si F_Π est vraie alors $time(Q, Y) = 2$

Corollaire 4.3.3. *Tant que F_Π est fausse :*

$$\tau_X^i(P)[Q] \star Y_i \succ_t \tau_X^{i+1}(P)[Q] \star Y_{i+1}$$

où :

1. $t = 2 + t_\Pi$
2. $Y_i = \tau_P^i(X) \sqcup P_\Pi^i(X)$
3. $P = P_{c_\Pi}^{F_\Pi}$ *le programme coquille*
4. $Q = \texttt{if } \neg F_\Pi \texttt{ then } P_\Pi \texttt{ else } \{i_{end} := i_{end} + 3;\}$ *le programme noyau*

Comme dit précédemment, nous n'utilisons pas uniquement le programme $P[Q]$ mais en fait le programme $P_\Pi\ P[Q]$. Ainsi le X dans le corollaire précédent est en fait $P_\Pi(X_0)$, où X_0 est l'état initial.

D'après le lemme p.94 nous avons que $\overline{F_\Pi}^{P_\Pi^i(P_\Pi(X_0))}$ devient vraie au bout de $i = time(\Pi, X_0|_{\mathcal{L}(\Pi)})$ répétitions de P_Π. Comme Q n'exécute alors plus P_Π, F_Π n'est plus mise à jour donc reste vraie [1] durant la fin de l'exécution.

Comme F_Π est vraie, après chaque étape de $\tau_X^i(P_{c_\Pi}^{F_\Pi})$ il faut $time(Q, Y) = 2$ étapes de Q pour faire la mise à jour $i_{end} := i_{end} + 3$. Comme cette variable est fraîche nous pouvons à nouveau séparer la mémoire en deux espaces distincts et itérer le lemme 4.3.2 :

Corollaire 4.3.4. *À partir du moment où F_Π devient vraie :*

$$\tau_X^i(P)[Q] \star \tau_P^i(X) \oplus (i_{end}, a) \succ_3 \tau_X^{i+1}(P)[Q] \star \tau_P^{i+1}(X) \oplus (i_{end}, a + 3)$$

où :

1. $P = P_{c_\Pi}^{F_\Pi}$ *le programme coquille*
2. $Q = \texttt{if } \neg F_\Pi \texttt{ then } P_\Pi \texttt{ else } \{i_{end} := i_{end} + 3;\}$ *le programme noyau*

D'après la définition 4.3.1, nous avons :

$$P[Q] = \{\} \text{ si et seulement si } P = \{\}$$

Or, d'après le corollaire 4.2.5 p.96, à partir du moment où F_Π devient vraie :

$$time(\tau_X^i(P), \tau_P^i(X)) \leq 2 \times (nest(P_{c_\Pi}) + 1)$$

Donc $\tau_X^i(P)[Q]$ fait au plus max_{end} étapes avant de s'arrêter, où :

$$max_{end} =_{def} 3 \times 2 \times (nest(P_{c_\Pi}) + 1)$$

max_{end} est ici une valeur, pas une variable. En revanche, la variable fraîche i_{end} étant initialisée à 0, nous aurons dans l'état final X_{final} :

$$i_{end} = 3 \times time(\tau_X^i(P), \tau_P^i(X))$$

où i est l'étape où F_Π est devenue vraie. Ainsi :

$$\overline{i_{end}}^{X_{final}} \leq max_{end}$$

TABLE 4.4 – Programme de « matelassage »

$$\begin{array}{l}
\texttt{if } i_{end} = max_{end}\ \{\}\ \texttt{else} \\
\quad \texttt{if } i_{end} = max_{end} - 1\ \{\}\ \texttt{else} \\
\qquad \ddots \\
\qquad\quad \texttt{if } i_{end} = 1\ \{\}\ \texttt{else} \\
\qquad\qquad \texttt{if } i_{end} = 0\ \{\}\ \texttt{else}\ \{\}
\end{array}$$

Il ne reste plus qu'à matelasser la fin de l'exécution en rajoutant à la fin de P_Π $P[Q]$ le programme $\texttt{skip}\ i_{end} \to max_{end} \in \texttt{While} \cap \texttt{LoopC}$, défini à la table 4.4 p.100.

Par récurrence sur $a \leq max_{end}$ nous avons pour tout Y :

$$time(\texttt{skip}\ i_{end} \leq max_{end}, Y \oplus (i_{end}, a)) = max_{end} - a + 1$$

Ainsi, quand F_Π devient vraie :

1. Le programme $\tau_X^i(P)[Q]$ fait i_{end} étapes,
2. puis $\texttt{skip}\ i_{end} \to max_{end}$ fait $max_{end} - i_{end} + 1$ étapes.

Donc quand F_Π devient vraie exactement $6 \times (nest(P_{c_\Pi}) + 1) + 1$ étapes sont faites avant la fin de l'exécution, qui ne dépend que de Π et plus de l'état initial.

Pour simuler l'ASM Π nous utiliserons donc le programme suivant :

$$P_\Pi\ P_{c_\Pi}^{F_\Pi}[\texttt{if } \neg F_\Pi \texttt{ then } P_\Pi \texttt{ else } \{i_{end} := i_{end} + 3;\}]\ \texttt{skip}\ i_{end} \to max_{end}$$

La simulation

Nous obtenons donc le théorème suivant :

Théorème 4.3.5. *(Simulation des ASMs par des programmes impératifs)*

$$\begin{array}{rl}
\texttt{While} & \textit{simule}\ \texttt{ASM} \\
\texttt{LoopC} & \textit{simule}\ \texttt{ASM}_{\mathcal{PR}} \\
\texttt{LoopC}_{\texttt{stat}} & \textit{simule}\ \texttt{ASM}_{\mathcal{P}ol}
\end{array}$$

Démonstration. Par construction p.92 il existe un programme P_{c_Π} tel que :

$$time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$$

et vérifiant selon la complexité c_Π de Π :

1. Si c_Π est quelconque alors $P_{c_\Pi} \in \texttt{While}$.
2. Si $c_\Pi \in \mathcal{PR}$ alors $P_{c_\Pi} \in \texttt{Loop}$.
3. Si $c_\Pi \in \mathcal{P}ol$ alors $P_{c_\Pi} \in \texttt{Loop}$, avec des bornes statiques et tel que :

$$depth(P_{c_\Pi}) = deg(c_\Pi)$$

1. ↑ En fait il aurait été possible de continuer à exécuter P_Π, puisque comme l'ASM ne fait plus de mises à jour F_Π resterait vraie. Mais comme nous utilisons de tout façon un test sur F_Π pour compter les étapes restantes nous avons préféré par lisibilité ne plus exécuter P_Π.

Remarque. Dans le cas $P_{c_\Pi} \in \texttt{While}$ aucune nouvelle variable n'est nécessaire.

Dans le cas $P_{c_\Pi} \in \texttt{Loop}$, que ce soit le programme de Meyer et Ritchie ou notre imbrication de boucles, les nouvelles variables ne dépendent que de c_Π.

Donc dans tous les cas $Var(P_{c_\Pi})$ ne dépend que de Π.

D'après la transformation 4.2.2 p.94 nous avons donc que :

1. Si c_Π est quelconque alors $P_{c_\Pi}^{F_\Pi} \in \texttt{While}$.
2. Si $c_\Pi \in \mathcal{PR}$ alors $P_{c_\Pi}^{F_\Pi} \in \texttt{LoopC}$.
3. Si $c_\Pi \in \mathcal{P}ol$ alors $P_{c_\Pi}^{F_\Pi} \in \texttt{LoopC}_{\texttt{stat}}$ tel que :

$$depth(P_{c_\Pi}^{F_\Pi}) = deg(c_\Pi)$$

Comme P_Π et $\texttt{skip}\ i_{end} \to max_{end}$ sont dans $\texttt{While} \cap \texttt{LoopC}$, c'est-à-dire qu'ils ne sont formés que de mises à jour et de conditionnelles, nous avons que le programme suivant :

$$P_\Pi\ P_{c_\Pi}^{F_\Pi}[\texttt{if}\ \neg F_\Pi\ \texttt{then}\ P_\Pi\ \texttt{else}\ \{i_{end} := i_{end} + 3;\}]\ \texttt{skip}\ i_{end} \to max_{end}$$

est (par composition p.48 et insertion p.98) dans le même langage que P_{c_Π}.

De plus, dans le cas $P_{c_\Pi}^{F_\Pi} \in \texttt{LoopC}_{\texttt{stat}}$ il possède la même profondeur, c'est-à-dire $deg(c_\Pi)$.

Par ces compositions et cette insertion nous avons ajouté au programme les variables $\{v_t \mid t \in T(\Pi)\}$ ainsi que i_{end}.

Donc les variables temporaires utilisées par la simulation ne dépendent bien que de Π et sont :

$$\boxed{Var(P_{c_\Pi}) \sqcup \{v_t \mid t \in T(\Pi)\} \sqcup \{i_{end}\}}$$

Jusqu'à ce que F_Π devienne vraie, l'exécution alterne entre :

1. t_Π étapes simulant une étape de Π
2. une étape de $P_{c_\Pi}^{F_\Pi}$
3. une étape $\texttt{if}\ \neg F_\Pi$, puis retour à 1.

Ainsi, chaque étape de Π est simulée par exactement $\boxed{d = t_\Pi + 2}$ étapes du programme. De plus l'exécution est assez longue car :

$$\text{Si } F_\Pi \text{ est fausse alors } time(P_{c_\Pi}^{F_\Pi}, X) \geq time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$$

Ainsi, $time(\Pi, X|_{\mathcal{L}(\Pi)})$ répétitions de ces trois étapes simulent l'ASM, puis d'après le lemme 4.2.1 p.94 t_Π étapes supplémentaires pour la dernière itération de P_Π rendent F_Π vraie [1].

Ensuite jusqu'à la fin de $P_{c_\Pi}^{F_\Pi}$ l'exécution alterne entre :

1. une étape de $P_{c_\Pi}^{F_\Pi}$
2. une étape $\texttt{if}\ \neg F_\Pi$
3. une étape $i_{end} := i_{end} + 3$, puis retour à 1.

Comme la variable i_{end} est initialisée à 0, à la fin de $P_{c_\Pi}^{F_\Pi}$ la valeur $\overline{i_{end}}^{X_{final}}$ est le nombre d'étapes faites depuis que F_Π est vraie.

D'après le corollaire 4.2.5 ce nombre d'étapes est borné par :

$$max_{end} = 6 \times (nest(P_{c_\Pi}) + 1)$$

1. ↑ Même si $time(\Pi, X|_{\mathcal{L}(\Pi)}) = 0$, auquel cas c'est le P_Π initial qui exécutera ces étapes.

Ensuite `skip` $i_{end} \rightarrow max_{end}$ est exécuté en $max_{end} - \overline{i_{end}}^{X_{final}} + 1$ étapes.
Donc la terminaison du programme nécessite exactement

$$t_{\Pi} + \overline{i_{end}}^{X_{final}} + max_{end} - \overline{i_{end}}^{X_{final}} + 1 \text{ étapes}$$

$$\text{Donc } \boxed{e = t_{\Pi} + 6 \times (nest(P_{c_{\Pi}}) + 1) + 1}$$

□

Chapitre 5

Conclusion

Sommaire

5.1 Le théorème 103
5.2 Critique 105
5.3 Perspectives 108

5.1 Le théorème

Rappelons que les algorithmes (séquentiels) sont définis p.22 comme étant les objets vérifiant les trois postulats :

1. Temps séquentiel (voir p.22)
2. États abstraits (voir p.24)
3. Exploration bornée (voir p.27)

Les classes en temps des algorithmes sont définies p.33, les langages impératifs sont définis p.46, et la restriction $\texttt{LoopC}_{\texttt{stat}}$ est définie p.81.

Théorème. *(Caractérisation impérative des classes algorithmiques)*

1. $\texttt{While} \simeq \texttt{Algo}$
2. $\texttt{LoopC} \simeq \texttt{Algo}_{\mathcal{PR}}$ *si les états sont* $\mathcal{PR}$*-space*
3. $\texttt{LoopC}_{\texttt{stat}} \simeq \texttt{Algo}_{\mathcal{P}ol}$ *avec* $depth(P) = deg(c_\Pi)$

Démonstration. Nous utilisons le théorème de Gurevich p.42 $\texttt{Algo} = \texttt{ASM}$ pour obtenir un modèle pour les algorithmes.

Nous prouvons l'équivalence algorithmique entre ces modèles par simulation mutuelle, définie p.56 :

- D'après le théorème 3.2.4 p.72 : ASM simule Imp.
  D'après la proposition 3.3.5 p.77 :
  Si les états sont $\mathcal{PR}$-space alors les programmes de LoopC sont $\mathcal{PR}$-time.
  D'après la proposition 3.4.6 p.82 :
  Les programmes de $\texttt{LoopC}_{\texttt{stat}}$ sont $\mathcal{P}ol$-time, avec $depth(P) = deg(\varphi_P)$.
  Donc :
  1. ASM simule While
  2. $\texttt{ASM}_{\mathcal{PR}}$ simule LoopC si les états sont $\mathcal{PR}$-space
  3. $\texttt{ASM}_{\mathcal{P}ol}$ simule $\texttt{LoopC}_{\texttt{stat}}$ avec $depth(P) = deg(c_\Pi)$

– Réciproquement, d'après le théorème 4.3.5 p.100 :

1. `While` simule `ASM`
2. `LoopC` simule $\texttt{ASM}_{\mathcal{PR}}$
3. $\texttt{LoopC}_{\texttt{stat}}$ simule $\texttt{ASM}_{\mathcal{P}ol}$ avec $depth(P) = deg(c_\Pi)$

□

Tous les animaux sont égaux, mais certains sont plus égaux que d'autres. — (George Orwell, *La Ferme des animaux*)

Remarque. Les ASMs simulent les programmes impératifs avec $d = 1$ et $e = 0$, alors que les programmes impératifs simulent les ASMs avec $d = t_\Pi + 2$ et $e = t_\Pi + 6 \times (nest(P_{c_\Pi}) + 1) + 1$, donc, bien que les modèles soient équivalents, les ASMs sont « plus équivalentes » que les programmes impératifs. À la section B.2 p.147 nous esquissons une extension $\texttt{Imp}^+$ de notre modèle `Imp` qui soit « aussi équivalente » que les ASMs.

De la même façon qu'il est possible d'utiliser le théorème de Gurevich p.42 pour établir que toute ASM peut se mettre sous une forme normale, il est possible d'utiliser notre théorème pour donner des formes normales aux programmes impératifs selon leur classe en temps :

Corollaire 5.1.1. *(Forme normale des programmes impératifs)*
Soit $P \in \texttt{Imp}$ et c_P sa classe de complexité (voir p.33).
Il existe un programme impératif $\tilde{P}$ équivalent à P (au sens de la définition 2.3.5 p.56) tel que (selon les cas) :

1. *Si c_P est quelconque alors $\tilde{P} = \{\texttt{while } F\ \{s\};\}$*
2. *Si $c_P \in \mathcal{PR}$ alors $\tilde{P} \in \texttt{LoopC}$ et il existe une constante $d \in \mathbb{N}^\star$ telle que pour tout $n \in Bound(\tilde{P})$ et pour toute exécution $\vec{X}$:*

$$\overline{n}^{X_i} \leq d \times i + max|X_0|$$

3. *Si $c_P \in \mathcal{P}ol$ alors $\tilde{P} \in \texttt{LoopC}_{\texttt{stat}}$ avec $depth(\tilde{P}) = deg(c_P)$*

Démonstration. Nous utilisons la transitivité (voir le lemme 2.3.6 p.56) de notre simulation. `Imp` est simulable par `ASM`, qui est simulable par :

1. `While` si c_P est quelconque (a priori $\mathcal{R}ec$ selon la remarque p.33).

   Entre les initialisations et le matelassage nous avons obtenu un programme de la forme $\{s_1; \texttt{while } F\ \{s_2\}; s_3\}$.

   Il est possible de le mettre sous la forme $\{\texttt{while } F\ \{s\};\}$ en utilisant deux variables *init* (initialisée à *true*) et *end* (initialisée à *false*), une dilatation temporelle $d = 4$ et un temps d'arrêt $e = 4$ avec le programme de la table 5.1 p.105.

2. `LoopC` si $c_P \in \mathcal{PR}$.

   Les bornes du programme font partie des variables $n_1, \ldots, n_k$ du programme $P_{c_P} \in$ `Loop`, et ne communiquent pas avec le reste de l'algorithme.

   Nous avons remarqué p.43 que la valeur d'une variable d'un programme de `Loop` est toujours bornée par la somme des valeurs initiales avec le nombre d'étapes déjà exécutées :

$$\overline{n_j}^{X_i} \leq \max_{1 \leq j \leq k} \overline{n_j}^{X_0} + i$$

TABLE 5.1 – Forme de Kleene

```
while (init = true ∨ end = false)
   if init = true then
      init := false; {s1}[skip 3]
   else if F then
      skip 1; {s2}[skip 3]
   else
      end := true; {s3}[skip 3]
```

Donc comme $\overline{n}^X + 1 = |\overline{n}^X|$ et que les variables de P_{c_P} sont des symboles de $\tilde{P}$, nous avons que pour tout $n \in Bound(\tilde{P})$:

$$\overline{n}^{X_i} \leq max|X_0| + i$$

Le coefficient d, lui, vient de la dilatation temporelle issue de la simulation. En effet, le programme P_{c_P} ne fait une étape que toutes les d étapes du programme $\tilde{P}$. D'où :

$$\overline{n}^{X_i} \leq d \times i + max|X_0|$$

3. $\texttt{LoopC}_{\texttt{stat}}$ si $c_P \in \mathcal{P}ol$, et nous avons montré que $depth(\tilde{P}) = deg(c_P)$.
   En particulier, pour tout $n \in Bound(\tilde{P}) \subseteq Stat(\tilde{P})$:

$$\overline{n}^{X_i} = \overline{n}^{X_0} \leq max|X_0|$$

□

La première partie du corollaire est une forme normale de Kleene, en revanche les deux dernières sont plus intrigantes :

Remarque. Il est notable que selon la classe en temps nous ayons une caractérisation de la taille des bornes en fonction de la durée écoulée :

1. $\overline{n}^{X_i} \leq a \times i + b$ pour un temps primitif récursif
2. $\overline{n}^{X_i} \leq c$ pour un temps polynomial

Il est cependant encore trop tôt pour savoir si un tel point de vue est pertinent...

5.2 Critique

QUISANI : Je trouve étrange que tu puisses prouver une équivalence algorithmique sans même préciser quelles sont les structures de données utilisées. Par exemple des opérations élémentaires comme l'addition ou la multiplication ne sont pas faites de la même façon selon que les entiers sont représentés en base unaire ou binaire.

AUTEUR : Mon résultat est à « structure de données » près, c'est-à-dire que si les ASMs disposent d'une structure de données je vais les simuler dans un modèle impératif en utilisant exactement les mêmes structures de données, et réciproquement. Il s'avère que les opérations utilisées n'importent que pour prouver que `LoopC` a un temps primitif récursif, contrairement au cas quelconque ou (de façon plus surprenante) au cas polynomial.

Q : Soit, mais ta présentation des structures de données est bien trop abstraite et ésotérique, et ne pourrait persuader des membres de la communauté travaillant sur le typage ou les contraintes d'espace.

A : À vrai dire c'est ce qui m'a motivé à rédiger la section B.1 p.139, bien qu'elle ne soit pas nécessaire à mon résultat. J'y développe notamment une formalisation du typage pour les ASMs, ainsi que la notion de représentation et de taille d'un élément.

Q : Je l'ai lue, et à vrai dire je trouve cette formalisation déraisonnable : il ne semble guère y avoir beaucoup de limites aux structures de données qui peuvent y être exprimées.

A : C'est justement ce qui fait sa force, et qui explique pourquoi Gurevich a supposé avec son second postulat que tout état était une structure du premier ordre. Si tu n'es pas à l'aise avec autant d'expressivité, j'ai esquissé un bestiaire des types usuels. Il est donc tout à fait possible de se restreindre uniquement aux algorithmes utilisant les types usuels. Mais je pense que la notion d'algorithme est intrinsèquement oraculaire, et qu'il ne faut donc pas présupposer des opérations disponibles.

Q : Admettons. J'aimerais revenir sur la définition de ta simulation p.56 : je veux bien admettre qu'il est commode de ne considérer que le comportement asymptotique d'un programme, mais malgré tout les constantes comptent vraiment dans des implémentations réalistes. Et d'ailleurs de nombreux articles sont consacrés à optimiser ces constantes, ou alors à trouver des versions plus optimales quand les entrées sont assez petites. En prenant une telle définition tu te retires de toi-même dans une tour d'ivoire de théoricien.

A : M'accordes-tu que les algorithmes auxquels tu penses vérifient les trois postulats ?

Q : Je te l'accorde, et ayant lu la preuve du théorème de Gurevich p.42 je suppose que tu veux te restreindre uniquement aux ASMs ?

A : En effet, ce théorème nous montre que les ASMs peuvent simuler étape par étape l'exécution d'un algorithme séquentiel, comme ceux que tu as cités. Dans ce cas si la complexité d'une ASM te dérange c'est que tu m'as donné à simuler un mauvais algorithme au départ.

Q : Ton argument est spécieux : peut-être que les ASMs simulent l'algorithme étape par étape, mais à cause de ta simulation il faut $d = r + c + m + 2$ étapes pour le simuler dans les programmes impératifs, où r est le nombre de termes lus par Π, c est le nombre d'états reconnus par Π et m est le degré de parallélisme de Π. Le parallélisme est le même que celui de l'algorithme de départ, et r est inférieur au nombre n de termes dans le témoin d'exploration. Mais c lui est de l'ordre de 2^n, ce qui est loin d'être négligeable comme constante !

A : En fait j'esquisse à l'exemple p.41 qu'en utilisant le fait que l'égalité soit une relation d'équivalence le nombre de conditionnelles possibles n'est pas 2^n mais plutôt le n-ième nombre de Bell, ce qui restreint les cas, mais j'admets que cela est du pinaillage. J'ai déjà esquissé l'idée que l'unité de temps était arbitraire, mais je comprends que cela ne semble guère convaincant face à des contraintes pratiques. C'est pour cela que j'ai esquissé dans la section B.2 p.147 un modèle $\texttt{Imp}^+$ « plus équivalent » aux ASMs.

Q : Si j'ai bien compris, cette extension du modèle initial utilise une parallélisation des mises à jours ainsi qu'une gratuité des conditionnelles et des boucles... c'est

en fait le même genre d'hypothèse que pour les ASMs elles-mêmes, n'est-ce pas ?

A : Exactement. Si cela est accepté pour les ASMs, pourquoi ne pas les accepter pour les langages impératifs ? Toutefois j'ai tenu à présenter un modèle « minimal » des langages impératifs afin que le résultat soit assez général, et notamment qu'il puisse s'appliquer aux langages de programmation usuels, comme le `C`, le `Python` ou le `Java`. Et donc j'ai dû poser une définition de l'équivalence algorithmique qui respecte cette dilatation temporelle.

Q : En lisant ton esquisse pour $\texttt{Imp}^+$ je suis tombé sur la remarque p.154, qui semble augurer un théorème de « speed-up ». C'est effectivement un argument solide pour la dilatation temporelle. Ce qui m'intrigue c'est que je me souviens qu'un tel théorème avait été obtenu pour les machines de Turing mais en changeant le langage, ce qui n'est pas le cas ici [1].

A : En effet, le langage et les états seraient les mêmes, mais je n'ose pas trop m'avancer pour l'instant puisque la preuve n'est pas encore faite. Toutefois, si le sujet t'intéresse la preuve dans le cas des machines de Turing consistait à grouper plusieurs symboles en un seul, par exemple utiliser un symbole par octets possible dans le cas binaire. Ce n'est d'ailleurs pas sans lien avec les machines de Turing à fenêtre développées dans [GV10].

Q : Entendu, merci. Mais je viens de penser à quelque chose. Dans cette section tu n'utilises pas la formule F_Π définie p.94 mais à la place un booléen b_{end}, qui me rappelle d'ailleurs celui que tu as utilisé pour obtenir une forme normale de Kleene p.105. Tu as écrit qu'il fallait que l'ASM soit sous forme normale, mais n'est-ce pas toujours le cas ?

A : Effectivement, il est toujours possible de le supposer à cause de la remarque p.42. Quand j'ai commencé à travailler sur la simulation je ne supposais pas que le programme était sous forme normale et c'est pour cela que j'ai réfléchi à la formule F_Π. Mais je ne voyais pas comment faire la preuve si l'ensemble de mises à jour faites par l'ASM n'était pas consistant ou admettait des mises à jour triviales, problèmes qui ne se posent pas avec la forme normale.

Q : Dans ce cas pourquoi ne pas avoir simplifié la présentation pour ne garder que le booléen, au lieu de cette formule immense ? En dehors de la fierté, bien sûr.

A : [*rire*] En dehors du fait que je la trouvais intéressante en elle-même, comme il est de toute façon nécessaire d'utiliser des variables temporaires pour paralléliser les mises à jour, cela ne coûte rien de les utiliser pour comparer l'état actuel et l'état précédent. Utiliser le booléen aurait coûté une variable supplémentaire, donc au final l'effort n'est pas perdu. C'est parce que ces variables temporaires ne sont pas nécessaires avec les n-uplets de $\texttt{Imp}^+$ que j'ai introduit b_{end}.

Q : C'est quand même assez choquant de considérer que F_Π soit évaluée en autant de temps que b_{end}, même si je sais que cela dérive directement du théorème de Gurevich. De la même façon ton esquisse de « speed-up » peut augmenter considérablement la taille des termes.

A : Je comprends tes réticences. Que voudrais-tu que je fasse ?

Q : Tu as défendu l'idée que les algorithmes étaient oraculaires : je propose de la prendre réellement au sérieux. Par exemple si je considère que mon modèle dispose d'un oracle me donnant l'addition de deux entiers, cela signifie que je peux calculer $x + y$ en une seule étape. Mais pour calculer $x + y + z$ il me

1. ↑ Il serait intéressant de comparer la preuve exacte avec les conditions de [BACS12] rendant possible ou non un théorème d'accélération linéaire.

faudra deux étapes et non une seule. Vu comme cela il n'est pas raisonnable que la mise à jour $r := x + y + z$ ne nécessite qu'une seule étape, et ce quelque soit le nombre d'additions faites. De la même façon s'il me faut une étape pour évaluer un simple booléen b_{end}, il devrait me falloir davantage d'étapes pour évaluer une grande formule comme F_Π .

A : C'est une idée intéressante. Dans ma formalisation, cela signifierait qu'une conditionnelle `if` F devrait être explorée non en 0 ou 1 étape, mais en $|F|$ étapes. De la même façon, une mise à jour $ft_1 \dots t_k := t_0$ devrait être évaluée en $1 + \sum_{0 \leq i \leq k} |t_i|$ étapes.

Q : Exactement : ne plus considérer les termes en bloc, mais compter le nombre de symboles dans les termes.

A : En fait il y a déjà une mesure de cela. Te rappelles-tu que dans le troisième postulat p.27 le témoin d'exploration T doit être clos par sous-termes ? Et bien s'il fallait évaluer $r := x + y$ le témoins d'exploration serait $\{r, x, y, x + y\}$ et non seulement $\{r, x + y\}$. Ainsi, en passant à $r := (x + y) + z$ le témoin d'exploration grossirait bien en devenant $\{r, x, y, z, x + y, (x + y) + z\}$: il y a bien autant de sous-termes que de symboles.

Q : Donc une étape de l'algorithme serait en fait $card(T)$ étapes élémentaires ? Dans la mesure où le témoin d'exploration caractérise les mises à jour faites entre deux états cela a du sens, oui... Mais attends, c'est un ensemble : que se passe-t-il dans le cas de la mise à jour $r := (x + x) + x$?

A : Oh, bien vu ! Et bien dans ce cas $T = \{r, x, x + x, (x + x) + x\}$. En un sens cela rappelle le procédé de mémoïsation... De plus, comme le témoin d'exploration est le même d'une étape à l'autre, le nombre $card(T)$ d'opérations élémentaires serait constant, et on en revient à la dilatation temporelle.

Q : À ceci près que tous les symboles ne sont pas utilisés à chaque étape, donc ce serait une borne mais elle ne devrait pas être toujours atteinte.

A : Tu sais, on peut toujours matelasser comme à la p.89...

Q : Justement, je croyais que ton but était d'avoir un langage de programmation où on pourrait écrire les meilleurs algorithmes. Pourquoi dans ce cas perdre du temps ?

A : Je l'ai fait uniquement pour prouver la condition de simulation la plus dure possible, afin que le résultat soit d'autant meilleur. Mais il est tout à fait possible de relâcher cette condition dans notre définition p.56. D'ailleurs je soupçonne que ce soit nécessaire pour des modèles de calcul où il n'existe pas au moins une variante de la commande `skip`.

Q : Je vois, merci pour tes réponses. Je n'ai pas d'autre question pour l'instant.

A : Merci à toi. À vrai dire, pour ma part il me reste quelques questions...

5.3 Perspectives

Nous avons souligné dans le dialogue p.5 que la démarche scientifique avait ceci d'original que nous cherchons surtout à prouver que nous avons tort. Une autre originalité est que chaque réponse a tendance à entraîner davantage de questions, ainsi notre ignorance semble s'accroître plus vite que nos connaissances. Mais ce que nous ignorons toujours aujourd'hui nous l'ignorions également hier. Ainsi, nous n'avons pas accru notre ignorance mais en avons davantage pris conscience.

> *Je suis plus sage que lui. Car il y a certes des chances qu'aucun de nous ne sache rien de beau ni rien de bon ; mais lui croit savoir quelque chose, alors qu'il ne sait rien, tandis que moi, si je ne sais rien, je ne crois pas non plus savoir.* — (Socrate[1], *Apologie de Socrate*)

Dans ce manuscrit nous nous sommes focalisés sur les preuves nécessaires à l'établissement de notre résultat, mais d'autres questions techniques sont apparues qui peuvent recevoir ici au moins une réponse partielle :

1. Simulation du `if`.

   Les langages `While` et `LoopC` disposent tous deux de boucles conditionnées. Dans un soucis de minimalité, on peut se demander si elles pourraient simuler une conditionnelle $\{\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}; s_3\}$. Une solution pour le `while` serait :

   $$\{\texttt{while}\ (F \wedge b)\ \{b := \mathit{false}; s_1\}; \texttt{while}\ (\neg F \wedge b)\ \{b := \mathit{false}; s_2\}; s_3\}$$

   Bien sûr, il faudrait respecter les conditions de notre simulation p.56, ce qui n'est pas le cas ici. En effet si F est vraie il faut deux étapes avant d'effectuer s_1 puis deux étapes avant d'effectuer s_3, alors que si F est fausse il faut trois étapes avant d'effectuer s_2 puis une étape avant d'effectuer s_3. Harmoniser le nombre d'étapes entre les deux cas n'est pas un problème, par exemple avec :

   $$\{\texttt{while}\ (F \wedge b)\ \{b := \mathit{false}; s_1\}; \texttt{while}\ (\neg F \wedge b)\ \{s_2; b := \mathit{false}\}; s_3\}$$

   Le problème est plutôt que $\{\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}; s_3\}$ fait une étape avec d'exécuter s_1 ou s_2 mais pas d'étape pour ensuite exécuter s_3. Ainsi, dans la simulation il nous faut payer des étapes pour sortir des boucles ou les effacer.

   Toutefois, le nombre de conditionnelles imbriquées étant borné dans chaque bloc considéré, il devrait être possible d'appliquer une méthode similaire à celle du corollaire 4.2.5 p.96 en bornant le nombre d'étapes par une constante ne dépendant que du programme.

   Nous nous en sommes toutefois abstenus par soucis de clarté et car les langages de programmation usuels ont toujours une version des conditionnelles.

   > *« toujours » et « jamais » sont deux mots qu'on devrait toujours se rappeler de ne jamais utiliser.* — (Alfred Korzybski)

2. Composition pour le `exit`.

   Dans [APV10] ce n'est pas `LoopC` qui est utilisé, mais le langage `LoopE` qui est le langage `Loop` (voir p.43) avec la commande `exit` suivante :

   $$\{\texttt{exit}; s\} \star X \succ \{\} \star X$$

   Dans [MV09] c'est bien `LoopC` qui est utilisé. Nous nous sommes d'ailleurs inspirés de l'argument montrant que `LoopC` est algorithmiquement équivalent à `LoopE` pour définir notre condition d'arrêt à la p.93.

   Dans ce manuscrit nous avons préféré utiliser `LoopC` essentiellement car il respecte les contextes donc facilite la preuve de la proposition 2.2.9 p.49, et notamment permet une composition simple des programmes.

1. ↑ Rapporté par Platon et traduit par Bernard et Renée Piettre.

Toutefois, il devait être possible de définir une composition de programme pour `LoopE`, et nous proposons le candidat suivant :

$$P_1P_2 =_{def} \{s_{P_1}[s_{P_2}; \texttt{exit}/\texttt{exit}]; s_{P_2}\}$$

où $s_{P_1}[s_{P_2}; \texttt{exit}/\texttt{exit}]$ est la séquence s_{P_1} où toutes les commandes `exit` sont remplacées par la séquence $s_{P_2}; \texttt{exit}$. En particulier, si P_1 contient une commande `exit` alors P_1P_2 fera bien les commandes de P_1 jusqu'au moment où il aurait dû sortir, puis fait les commandes de P_2 avant de sortir. Sinon P_1 ne contient pas de commande `exit` et dans ce cas $\{s_{P_1}[s_{P_2}; \texttt{exit}/\texttt{exit}]; s_{P_2}\} = \{s_{P_1}; s_{P_2}\}$ de façon plus usuelle.
Ainsi, notre preuve devrait pouvoir être refaite en remplaçant `LoopC` par `LoopE`. Mais cela n'est pas nécessaire puisqu'en utilisant la transitivité de la simulation (voir p.56) nous pouvons la déduire de notre résultat et de l'équivalence entre `LoopC` et `LoopE`.

3. Variantes des boucles.
   Nous avons déjà évoqué à la p.46 qu'il était équivalent de lire ou non le compteur de le corps d'une boucle, puisque la commande :

   $$\texttt{for}(j = 0; j < n; j{+}{+})\ \{s[j]\}$$

   peut être simulée par :

   $$j := 0; \texttt{loop}\ n\ \{s[j]; j := j + 1\}$$

   Il faut toutefois prendre la précaution de rajouter des `skip` pour les autres commandes afin d'homogénéiser la dilatation temporelle.
   Dans certains modèles c'est une boucle sur une liste (voir p.144 pour notre présentation des listes) qui est plus naturelle, et une commande :

   $$\texttt{for}\ a\ \texttt{in}\ \ell\ \{s[a]\}$$

   peut être simulée de façon similaire par :

   $$\ell' := \ell; \texttt{loop}\ \mathit{length}(\ell)\ \{a := \mathit{head}(\ell'); \ell' := \mathit{tail}(\ell'); s[a]\}$$

   Si *length* n'est pas disponible dans le langage, comme $|\ell| \geq \mathit{length}(\ell)$ il est possible d'utiliser la variante :

   $$\ell' := \ell; \texttt{loop}\ |\ell|\ \texttt{except}\ \ell' = ()\ \{a := \mathit{head}(\ell'); \ell' := \mathit{tail}(\ell'); s[a]\}$$

   Ainsi, du moment qu'il est possible d'arrêter le schéma de récursion en cours d'exécution une variété de schémas semble possible.

Il ne reste à présent que les questions moins techniques mais plus profondes. Obtenir des réponses nécessiterait probablement un autre manuscrit, aussi avant de clore celui-ci nous nous contenterons de les esquisser.

4. Les récurseurs de Moschovakis.
   Lors de notre dialogue introductif nous avons esquissé la définition des algorithmes selon Moschovakis ainsi que la divergence de point de vue avec la présentation des ASMs de Gurevich. En dehors de la différence de présentation, paradigme impératif contre fonctionnel, il y a une différence d'approche plus profonde sur laquelle nous avons trouvé intéressant de revenir, au moins brièvement.
   Dans son article [Mos01] Moschovakis définit les algorithmes comme des « récurseurs », c'est-à-dire un système d'équations récursives admettant un point fixe, ce point fixe étant assuré par la continuité des fonctions considérées. Sa critique d'un modèle mécanique prend deux formes :

a. Il prend l'exemple du *mergesort* pour illustrer qu'une implémentation mécanique d'un algorithme doit se faire en précisant tous les choix faits. Notamment comment est formalisée la pile représentant les appels récursifs, comment est faite la substitution, ou quel est l'ordre d'évaluation. Pourtant ces choix ne sont pas pertinents dans la preuve que l'algorithme calcule bien ce qui est attendu et combien d'opérations élémentaires sont faites : seules les équations définissant l'algorithme importent. Ainsi, ces détails d'implémentation ne devraient pas être considérés comme faisant partie de l'algorithme.

Gurevich répond dans [BG02] que ces équations ne sont que des spécifications, dont l'interprétation n'est ni claire ni forcément faisable, et que certains choix sont faits malgré tout pour obtenir l'estimation de Moschovakis. Gurevich propose également le modèle des ASMs récursives [GS97], un modèle particulier des ASMs distribuées, pour implémenter fidèlement l'exemple du *mergesort*. Toutefois, la question de savoir si cet exemple peut être implémenté naturellement [1] uniquement dans le cadre des trois postulats reste entière, même s'il existe des algorithmes récursifs se traduisant simplement sous forme itérative, comme les tours de Hanoï.

À titre de comparaison voici la fonction d'Ackermann sous forme récursive :

$$\begin{cases} ack(0,n) = n+1 \\ ack(m+1,0) = ack(m,1) \\ ack(m+1,n+1) = ack(m, ack(m+1,n)) \end{cases}$$

Et en voici une forme itérative avec une liste [2] ℓ à gauche :

$$\begin{array}{rcr} \ell, 0 \mid n & \succ & \ell \mid n+1 \\ \ell, m+1 \mid 0 & \succ & \ell, m \mid 1 \\ \ell, m+1 \mid n+1 & \succ & \ell, m, m+1 \mid n \end{array}$$

b. Les preuves faites pour établir les récurseurs n'utilisent en réalité pas la continuité : la monotonie suffit. Ainsi, il serait légitime de considérer également des algorithmes non continus, qui selon Moschovakis ne pourraient être légitimement implémentables en raison de leur nature profondément infinitaire. Par exemple il serait possible d'affirmer s'il existe des témoins vérifiant une propriété R quelconque, même sur un domaine infini.

L'exemple donné par Moschovakis rappelle les problèmes de constructivité que nous avons rencontrés dans des versions antérieures de notre présentation, avant de restreindre les formules de Gurevich (voir p.19) au cas sans quantificateur :

$$\texttt{if } \exists x\ Rx \texttt{ then } r := \mathit{true} \texttt{ else } r := \mathit{false}$$

Nous n'avons toutefois pas inclus les quantificateurs à notre présentation, d'une part parce que les formules rencontrées sous forme normale p.40 n'en contiennent pas, et d'autre part parce qu'un programme de la forme :

$$\texttt{if } \forall x\ \exists y_x\ F[x, y_x] \texttt{ then } r := y_t \texttt{ else } r := \mathit{undef}$$

1. ↑ L'utilisation d'une pile étant alors discutable puisque ne faisant pas partie de la structure de données initiale.

2. ↑ Inversée pour correspondre visuellement davantage aux équations. La machine s'arrête quand la liste à gauche est vide.

consiste si possible à fabriquer un y_t vérifiant $F[t, y_t]$ pour un terme t donné, ce qui revient à disposer implicitement d'une nouvelle opération n'étant pas déclarée explicitement dans le langage. Nous estimons que cela contredit l'oracularité, principe nécessaire à une présentation raisonnable des algorithmes.

En l'état actuel de nos connaissances il ne nous semble donc pas clair [1] de trancher entre les deux approches, partant de points de vue (machine contre récursion) mais aussi de formalismes (structures du premier ordre contre domaines de Scott) assez différents quoique intéressants.

5. Place de nos travaux dans la hiérarchie de complexité.

   Notre intérêt pour la classe $\mathtt{ASM}_{\mathcal{P}ol}$ en général et pour une caractérisation syntaxique des degrés en particulier vient de notre préoccupation pour la faisabilité. C'est d'ailleurs cette préoccupation qui nous a conduit à nous préoccuper p.143 de savoir si nous pouvions ou non nous restreindre à des structures de données usuelles.

   Dans cette optique, situer nos résultats dans le cadre de la hiérarchie de complexité usuelle paraît non seulement désirable mais nécessaire. `While` et `LoopC` sont clairement au-delà de `EXPTIME`, mais pas $\mathtt{LoopC}_{\mathtt{stat}}$ qui calcule de façon déterministe en un temps polynomial, donc semble correspondre à la classe `P`. Il faut néanmoins rester prudent, car nous avons utilisé des oracles en guise de fonction élémentaire.

   La possibilité abordée un peu plus haut d'utiliser des quantificateurs dans les formules n'est pas sans rappeler les systèmes de preuve interactifs, notamment le théorème de Shamir $\mathtt{IP} = \mathtt{PSpace}$ [Sha92], ce qui est amusant car nous n'avons fait aucune hypothèse sur l'espace disponible de $\mathtt{LoopC}_{\mathtt{stat}}$. En passant, si le système de preuve interactif est déterministe alors $\mathtt{dIP} = \mathtt{NP}$, ce qui renforce l'idée annoncée p.22 que le non-déterminisme peut être vu comme une interaction avec l'environnement.

   Ainsi, il pourrait être intéressant de chercher réellement à caractériser $\mathtt{LoopC}_{\mathtt{stat}}$ dans la hiérarchie de complexité, ainsi que d'étudier les variations selon les oracles ou l'espace disponible, mais aussi si les quantificateurs ou le non-déterminisme étaient introduits.

En plus de ces questions à explorer, il reste encore beaucoup de travail à faire, mais que nous espérons réaliser dans le futur. Notamment :

6. Implémentation ASM de stratégies du lambda-calcul.

   L'exemple p.27 ainsi que la version itérative de la fonction d'Ackermann sont issus du questionnement sur le lien entre les présentations de Gurevich et Moschovakis, notamment si une présentation récursive est équivalente à une présentation mécanique. Dans cet optique nous avons commencé à étudier une implémentation ASM de certaines stratégies d'évaluation du lambda-calcul (par nom, par valeur et par besoin), qui a renforcé notre impression p.73 que les ASMs pouvaient être vues non comme un modèle de calcul mais comme un modèle pour les processeurs.

7. Preuve de $\mathtt{Imp}^+ \simeq \mathtt{ASM}$ et de l'accélération liénaire.

   Nous avons esquissé à la section B.2 p.147 le modèle de calcul $\mathtt{Imp}^+$ pour obtenir une simulation plus équitable avec les ASMs. Nous pensons que pour finir la preuve il suffirait de définir pour les chemins entre deux mises à jour un objet qui serait similaire à ce que nos graphes d'exécutions p.63 sont aux numéros de ligne pour les programmes. Toutefois, à dilatation temporelle près notre résultat avec `Imp` convient, donc ce n'est pas forcément nécessaire.

1. ↑ Soare a toutefois fait dans [Soa96] un travail de compilation remarquable autour de la distinction entre récursivité et calculabilité, qui aide grandement à clarifier les idées.

Montrer un théorème d'accélération linéaire pour les ASMs renforcerait cette idée, mais ne serait pas suffisant. En effet, pour l'instant une étape de Π est simulée en $d > 0$ étapes avec un programme P. S'il était possible, en suivant une méthode similaire à celle de la p.153, d'obtenir un programme d'ASM Π^d qui soit d fois plus rapide que Π nous aurions :

$$\tau^i_{\Pi^d}(X) = \tau^{d\times i}_{\Pi}(X)$$

Si, en suivant cette méthode, le programme Π^d avait les mêmes caractéristiques r, c et m (voir la proposition p.89) que celles de Π, il faudrait la même dilatation temporelle d pour simuler Π^d avec un programme P^d :

$$\tau^{d\times i}_{P^d}(X) = \tau^i_{\Pi^d}(X)$$

Donc nous aurions $\tau^{d\times i}_{P^d}(X) = \tau^{d\times i}_{\Pi}(X)$ et non $\tau^i_{P^d}(X) = \tau^i_{\Pi}(X)$.
Dit autrement, si on peut simuler une étape avec d étapes et qu'on applique une accélération linéaire de d étapes alors on simule d étapes en d étapes et non une étape en une étape. Ainsi, cela nous rapprocherait de l'identité entre algorithmes définie p.23 mais nous ne l'aurions pas encore atteinte [1].

8. Développer $\texttt{LoopC}_{\texttt{stat}}$.

   Nous avons été surpris d'obtenir une caractérisation aussi simple, et même ne tenant pas compte de l'espace, des algorithmes polynomiaux, surtout en comparaison avec les travaux de Neergaard [Nee03]. De plus, nous avons obtenu une caractérisation purement syntaxique du degré du polynôme, ce qui permet de se restreindre à des algorithmes « faisables » : par exemple, il est difficile de considérer un algorithme en $O(n^{42})$ comme tel.

   Nous espérons, avec Pierre Valarcher, développer des exemples plus concrets et traduire notre résultat sous forme de schéma de récursion. Cela serait l'occasion de faire le lien avec d'autres modèles de calcul, par exemple [BGM10] qui nous a intéressé notamment car il utilise une notion de fonction basique, une présentation de la taille similaire à notre présentation, et un schéma de récursion assez large pour obtenir le minimum p.24 avec la bonne complexité.

   Nous souhaitons donc non seulement participer à la quête d'un modèle de calcul pour l'ensemble des algorithmes polynomiaux mais aussi, si les trois postulats sont acceptés, présenter un modèle candidat pour la clore.

1. ↑ Cela étant dit, même si une simulation est strictement étape par étape il reste le problème des variables temporaires.

Annexe A

Lemmes techniques

Sommaire

A.1 Logique du premier ordre **115**
Clôture par sous-termes 115
Isomorphisme de structures 117
Restriction de structure 119
A.2 Les programmes impératifs **121**
Composition de programmes 121
Mises à jour d'un programme impératif 124
A.3 Graphes d'exécution **127**
Finitude d'un graphe d'exécution 127
Composition de graphes d'exécution 128
Sous-graphes d'exécution 129
A.4 Traduction des ASMs **130**
Traduction d'un programme en forme normale 130
Condition d'arrêt 133
Sémantique de l'insertion de programme 135

A.1 Logique du premier ordre

Clôture par sous-termes

Définition A.1.1. (Clôture par sous-termes)
L'ensemble $Sub(t)$ des sous-termes de t est défini par induction sur t :

1. Si $t = c \in F_0(\mathcal{L})$
   Alors $Sub(t) =_{def} \{c\}$
2. Si $t = ft_1 \dots t_\alpha$
   où $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et $t_1, \dots, t_\alpha \in Term(\mathcal{L})$
   Alors $Sub(t) =_{def} \{t\} \cup \bigcup_{1 \leq i \leq \alpha} Sub(t_i)$

Si T est un ensemble de termes, soit $Sub(T) =_{def} \bigcup_{t \in T} Sub(t)$.

Comme $t \in Sub(t)$, nous avons $T \subseteq \bigcup_{t \in T} Sub(t) = Sub(T)$

Si $Sub(T) = T$ nous dirons que T est **clos par sous-termes**.

Lemme A.1.2. *Si $t \in Sub(t')$ alors $Sub(t) \subseteq Sub(t')$.*

Démonstration. Par induction sur t' :

$\boldsymbol{t' = c \in F_0(\mathcal{L})}$

Dans ce cas $Sub(t') = \{c\}$ donc $t = c = t'$

D'où $Sub(t) = Sub(t')$

$\boldsymbol{t' = ft_1 \dots t_\alpha}$

où $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et $t_1, \dots, t_\alpha \in Term(\mathcal{L})$

Dans ce cas $Sub(t') =_{def} \{t'\} \cup \bigcup_{1 \leq i \leq \alpha} Sub(t_i)$

Si $t = t'$ alors $Sub(t) = Sub(t')$

Sinon il existe $1 \leq i \leq \alpha$ tel que $t \in Sub(t_i)$.

D'après l'hypothèse d'induction : $Sub(t) \subseteq Sub(t_i)$

Or : $Sub(t_i) \subseteq Sub(t')$

Donc : $Sub(t) \subseteq Sub(t')$

□

Lemme A.1.3. *$Sub(T)$ est clos par sous-termes.*

Démonstration. Supposons $t \in Sub(Sub(T)) = \bigcup_{t' \in Sub(T)} Sub(t')$.

Dans ce cas il existe $t' \in Sub(T)$ tel que $t \in Sub(t')$.

Comme $t' \in Sub(T) = \bigcup_{t'' \in T} Sub(t'')$

Il existe $t'' \in T$ tel que $t' \in Sub(t'')$

D'où d'après le lemme A.1.2 : $Sub(t') \subseteq Sub(t'')$

D'où $t \in Sub(t'')$, avec $t'' \in T$

Donc $t \in \bigcup_{t'' \in T} Sub(t'') = Sub(T)$

Et donc : $Sub(Sub(T)) \subseteq Sub(T)$

De plus $Sub(T) \subseteq Sub(Sub(T))$

D'où l'égalité. □

Lemme A.1.4. *Si T est fini alors $Sub(T)$ est fini.*

Démonstration. Montrons par induction sur t que $Sub(t)$ est fini :

$\boldsymbol{t = c}$

Dans ce cas $Sub(t) = \{c\}$ est fini.

$\boldsymbol{t = ft_1 \dots t_\alpha}$

Dans ce cas $Sub(t) = \{t\} \cup \bigcup_{1 \leq i \leq \alpha} Sub(t_i)$.

D'après l'hypothèse d'induction les $Sub(t_i)$ sont finis, et la réunion est finie.

Donc $Sub(t)$ est fini.

Si T est fini alors la réunion $Sub(T) = \bigcup_{t \in T} Sub(t)$ est finie.

Or les $Sub(t)$ sont finis.

Donc $Sub(T)$ est fini aussi. □

Isomorphisme de structures

Lemme A.1.5. *Si ζ est un isomorphisme entre X et Y, alors ζ est injective.*

Démonstration. Supposons $\zeta(a) = \zeta(b)$ et montrons par l'absurde que $a = b$:
En appliquant l'isomorphisme à l'égalité, nous avons :
– $\zeta(\overline{\chi_=}^X(a,b)) = \overline{\chi_=}^Y(\zeta(a), \zeta(b))$
– $\zeta(\overline{false}^X) = \overline{false}^Y$
Or, si $a \neq b$ alors $=^X(a,b) = \overline{false}^X$.
D'où $\overline{\chi_=}^Y(\zeta(a), \zeta(b)) = \zeta(\overline{\chi_=}^X(a,b)) = \zeta(\overline{false}^X) = \overline{false}^Y$
Mais comme $\zeta(a) = \zeta(b)$ nous avons $\overline{\chi_=}^Y(\zeta(a), \zeta(b)) = \overline{true}^Y$
Ce qui contredit $\overline{true}^Y \neq \overline{false}^Y$.
Donc $a = b$. □

Prouvons à présent nos remarques p.21 :

Lemme A.1.6. *Si ζ est un isomorphisme entre X et Y*
alors F est vraie (resp. fausse) sur $X \Leftrightarrow F$ est vraie (resp. fausse) sur Y

Démonstration. Rappelons que les formules sont définies p.19 :

$$\begin{aligned} F \text{ est vraie dans } X &\Leftrightarrow \overline{F}^X = \overline{true}^X \\ &\Leftrightarrow \zeta(\overline{F}^X) = \zeta(\overline{true}^X) \\ &\Leftrightarrow \overline{F}^Y = \overline{true}^Y \\ &\Leftrightarrow F \text{ est vraie dans } Y \end{aligned}$$

Et de même pour F est fausse sur $X \Leftrightarrow F$ est fausse sur Y, en remplaçant *true* par *false*. □

La définition suivante est donnée dans Cori-Lascar [CL03a], et nous la prendrons comme définition « classique » de l'isomorphisme entre structures :

Définition A.1.7. (Définition « classique » de l'isomorphisme)
ζ est un isomorphisme entre X et Y si :

1. ζ est surjective
2. Pour tout $c \in F_0(\mathcal{L})$: $\zeta(\overline{c}^X) = \overline{c}^Y$
3. Pour tout $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et pour tout $a_1, \ldots, a_\alpha \in \mathcal{U}(X)$:
   $\zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) = \overline{f}^Y(\zeta(a_1), \ldots, \zeta(a_\alpha))$
4. Pour tout $R \in R_\alpha(\mathcal{L})$ et pour tout $a_1, \ldots, a_\alpha \in X$
   $(a_1, \ldots, a_\alpha) \in \overline{R}^X$ si et seulement si $(\zeta(a_1), \ldots, \zeta(a_\alpha)) \in \overline{R}^Y$

Lemme A.1.8. *La définition 1.1.9 p.20 est équivalente à la définition A.1.7.*

Démonstration. En effet, par définition p.18 de χ_R :
– $\vec{a} \in \overline{R}^X \Leftrightarrow \overline{\chi_R}^X(\vec{a}) = \overline{true}^X$
– $\overrightarrow{\zeta(a)} \in \overline{R}^Y \Leftrightarrow \overline{\chi_R}^Y(\overrightarrow{\zeta(a)}) = \overline{true}^Y$
Par 3. nous avons :
– $\zeta(\overline{\chi_R}^X(\vec{a})) = \overline{\chi_R}^Y(\overrightarrow{\zeta(a)})$
– $\zeta(\overline{true}^X) = \overline{true}^Y$

Donc $\overrightarrow{\zeta(a)} \in \overline{R}^Y \Leftrightarrow \zeta(\overline{\chi_R}^X(\vec{a})) = \zeta(\overline{true}^X)$
Or d'après le lemme A.1.5 ζ est une application injective, donc :
$\zeta(\overline{\chi_R}^X(\vec{a})) = \zeta(\overline{true}^X) \Leftrightarrow \overline{\chi_R}^X(\vec{a}) = \overline{true}^X$
D'où : $\vec{a} \in \overline{R}^X \Leftrightarrow \overrightarrow{\zeta(a)} \in \overline{R}^Y$ □

Lemme (1.1.10 p.21). *(Inverse d'un isomorphisme)*
Si ζ est un isomorphisme entre X et Y
alors ζ^{-1} est un isomorphisme entre Y et X.

Démonstration. Comme ζ est injective et surjective, elle est bijective, et nous pouvons définir ζ^{-1}. Nous avons :

1. $\zeta^{-1} : \mathcal{U}(Y) \to \mathcal{U}(X)$ est une application surjective
2. Pour tout $c \in F_0(\mathcal{L})$:
   $\zeta^{-1}(\overline{c}^Y) = \zeta^{-1}(\zeta(\overline{c}^X)) = \overline{c}^X$
3. Pour tout $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et pour tout $a_1, \ldots, a_\alpha \in \mathcal{U}(X)$:

$$\begin{aligned}\zeta^{-1}(\overline{f}^Y(b_1, \ldots, b_\alpha)) &= \zeta^{-1}(\overline{f}^Y(\zeta(a_1), \ldots, \zeta(a_\alpha))) \\ &= \zeta^{-1}(\zeta(\overline{f}^X(a_1, \ldots, a_\alpha))) \\ &= \overline{f}^X(a_1, \ldots, a_\alpha) \\ &= \overline{f}^X(\zeta^{-1}(b_1), \ldots, \zeta^{-1}(b_\alpha))\end{aligned}$$

Donc ζ^{-1} est un isomorphisme entre Y et X. □

En utilisant la définition p.21 nous montrons que :

Lemme (1.1.12 p.21). *(Isomorphisme par remplacement)*
Si $B \cap \mathcal{U}(X) = \varnothing$ alors la structure Y obtenue en remplaçant dans X les éléments de A par les éléments de B est isomorphe à X.
De plus, s'il existe un ensemble de termes clos T clos par sous-termes et tel que $\forall t \in T\ \overline{t}^X \notin A$ alors $\forall t \in T\ \overline{t}^X = \overline{t}^Y$.

Démonstration. Soit $\zeta : \mathcal{U}(X) \to \mathcal{U}(Y)$ définie par :
- $\zeta(a) = \varphi(a)$ si $a \in A$
- $\zeta(a) = a$ sinon

Montrons que $\zeta : X \to Y$ est un isomorphisme :

1. Pour tout $b \in Y$:
   - Si $b \in B$ il existe $a \in A$ tel que $\varphi(a) = b$ d'où $\zeta(a) = \varphi(a) = b$
   - sinon $b \in \mathcal{U}(X) \setminus A$ d'où $\zeta(b) = b$

   D'où ζ est surjective.
2. Si $c \in F_0(\mathcal{L})$
   $\zeta(\overline{c}^X) = \varphi(\overline{c}^X)$ si $\overline{c}^X \in A$
   et $\zeta(\overline{c}^X) = \overline{c}^X$ sinon
   D'où : $\zeta(\overline{c}^X) = \overline{c}^Y$
3. Si $f \in F_\alpha(\mathcal{L})$
   Soit $a \in \mathcal{U}(X)$.
   a. Si $a \in A$ alors $\zeta(a) = \varphi(a) \in B$
      D'où : $\zeta(a)' = \varphi^{-1}(\zeta(a)) = \varphi^{-1}(\varphi(a)) = \mathrm{a}$

b. Si $a \notin A$ alors $\zeta(a) = a$.
Si $\zeta(a) \in B$ alors $a \in B$
Donc $B \cap \mathcal{U}(X) \neq \varnothing$, ce qui contredit l'hypothèse.
D'où $\zeta(a) \notin B$ et donc $\zeta(a)' = \zeta(a) = a$.

Dans tous les cas : $\zeta(a)' = a$.
D'où si $a_1, \ldots, a_\alpha \in \mathcal{U}(X)$
Alors $\zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) = \zeta(\overline{f}^X(\zeta(a_1)', \ldots, \zeta(a_\alpha)'))$

a. si $\overline{f}^X(\zeta(a_1)', \ldots, \zeta(a_\alpha)') \in A$
alors $\zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) = \varphi(\overline{f}^X(\zeta(a_1)', \ldots, \zeta(a_\alpha)'))$

b. si $\overline{f}^X(\zeta(a_1)', \ldots, \zeta(a_\alpha)') \notin A$
alors $\zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) = \overline{f}^X(\zeta(a_1)', \ldots, \zeta(a_\alpha)')$

D'où $\zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) = \overline{f}^Y(\zeta(a_1), \ldots, \zeta(a_\alpha))$.

La dernière partie du lemme se prouve par induction sur $t \in T$:

$\boldsymbol{t = c \in F_0(\mathcal{L})}$
Comme $\overline{c}^X = \overline{t}^X \notin A$, nous avons $\overline{c}^Y = \overline{c}^X$
D'où $\overline{t}^Y = \overline{t}^X$.

$\boldsymbol{t = f t_1 \ldots t_\alpha}$
Où $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et $t_1, \ldots, t_\alpha$ sont des termes.
T est clos par sous-termes, d'où pour tout $1 \leq i \leq \alpha$: $t_i \in T$
Par hypothèse sur T : $\overline{t_i}^X \notin A$, d'où $\overline{t_i}^X \in \mathcal{U}(X) \setminus A$
Donc $\overline{t_i}^X \in \mathcal{U}(Y)$ et on peut définir $(\overline{t_i}^X)'$.
Comme $\overline{t_i}^X \in \mathcal{U}(Y)$, $\overline{t_i}^X \in B$ contredirait $B \cap \mathcal{U}(X) = \varnothing$.
Donc $\overline{t_i}^X \notin B$ et donc $(\overline{t_i}^X)' = \overline{t_i}^X$, d'où :

$$\begin{aligned}
\overline{f}^X((\overline{t_1}^X)', \ldots, (\overline{t_\alpha}^X)') &= \overline{f}^X(\overline{t_1}^X, \ldots, \overline{t_\alpha}^X) \\
&= \overline{t}^X \notin A \text{ par hypothèse sur } T \\
\text{d'où } \overline{f}^Y(\overline{t_1}^X, \ldots, \overline{t_\alpha}^X) &= \overline{f}^X((\overline{t_1}^X)', \ldots, (\overline{t_\alpha}^X)') \\
&= \overline{t}^X
\end{aligned}$$

Comme $t_i \in T$, d'après l'hypothèse d'induction : $\overline{t_i}^X = \overline{t_i}^Y$, d'où :

$$\begin{aligned}
\overline{t}^Y &= \overline{f}^Y(\overline{t_1}^Y, \ldots, \overline{t_\alpha}^Y) \\
&= \overline{f}^Y(\overline{t_1}^X, \ldots, \overline{t_\alpha}^X) \\
&= \overline{t}^X
\end{aligned}$$

□

Restriction de structure

Lemme (**2.3.3** p.54)**.** *(Isomorphisme par restriction)*
Soient :

1. *$\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages tels que $\mathcal{L}_1 \supseteq \mathcal{L}_2$*
2. *X et Y deux $\mathcal{L}_1$-structures*

Si X et Y sont isomorphes alors $X|_{\mathcal{L}_2}$ et $Y|_{\mathcal{L}_2}$ le sont aussi.

Démonstration. Soit ζ l'isomorphisme entre X et Y :

1. La restriction respecte les univers donc ζ reste surjective.
2. Si $c \in F_0(\mathcal{L}_2)$ alors $\zeta(\overline{c}^{X|_{\mathcal{L}_2}}) = \zeta(\overline{c}^X) = \overline{c}^Y = \overline{c}^{Y|_{\mathcal{L}_2}}$
3. Si $f \in F_\alpha(\mathcal{L}_2)$ avec $\alpha > 0$ alors :

$$\begin{aligned}\zeta(\overline{f}^{X|_{\mathcal{L}_2}}(a_1, \ldots, a_\alpha)) &= \zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) \\ &= \overline{f}^Y(\zeta(a_1), \ldots, \zeta(a_\alpha)) \\ &= \overline{f}^{Y|_{\mathcal{L}_2}}(\zeta(a_1), \ldots, \zeta(a_\alpha))\end{aligned}$$

Donc ζ est aussi un isomorphisme entre $X|_{\mathcal{L}_2}$ et $Y|_{\mathcal{L}_2}$. □

Rappel. $\Delta|_{\mathcal{L}} =_{def} \{(f, \vec{a}, b) \in \Delta \mid f \in \mathcal{L}\}$

Lemme A.1.9. *Si $\Delta = \Delta_1 \cup \Delta_2$ alors $\Delta|_{\mathcal{L}} = \Delta_1|_{\mathcal{L}} \cup \Delta_2|_{\mathcal{L}}$*

Démonstration.

$$\begin{aligned}\Delta|_{\mathcal{L}} &= (\Delta_1 \cup \Delta_2)|_{\mathcal{L}} \\ &= \{(f, \vec{a}, b) \in \Delta_1 \cup \Delta_2 \mid f \in \mathcal{L}\} \\ &= \{(f, \vec{a}, b) \in \Delta_1 \mid f \in \mathcal{L}\} \cup \{(f, \vec{a}, b) \in \Delta_2 \mid f \in \mathcal{L}\} \\ &= \Delta_1|_{\mathcal{L}} \cup \Delta_2|_{\mathcal{L}}\end{aligned}$$

□

Lemme A.1.10. *Si Δ est consistant alors $(X \oplus \Delta)|_{\mathcal{L}} = X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}}$*

Démonstration. Comparons ces deux structures :

1. $\mathcal{L}((X \oplus \Delta)|_{\mathcal{L}}) = \mathcal{L}$
   et $\mathcal{L}(X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}}) = \mathcal{L}(X|_{\mathcal{L}}) = \mathcal{L}$
   Donc $\mathcal{L}((X \oplus \Delta)|_{\mathcal{L}}) = \mathcal{L}(X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}})$
2. $\mathcal{U}((X \oplus \Delta)|_{\mathcal{L}}) = \mathcal{U}(X \oplus \Delta) = \mathcal{U}(X)$
   et $\mathcal{U}(X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}}) = \mathcal{U}(X|_{\mathcal{L}}) = \mathcal{U}(X)$
   Donc $\mathcal{U}((X \oplus \Delta)|_{\mathcal{L}}) = \mathcal{U}(X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}})$
3. Comme $\Delta|_{\mathcal{L}} \subseteq \Delta$ et que Δ is consistent, $\Delta|_{\mathcal{L}}$ est consistant aussi.
   Soit $f \in \mathcal{L}$.
   Nous avons $(f, \vec{a}, b) \in \Delta \Leftrightarrow (f, \vec{a}, b) \in \Delta|_{\mathcal{L}}$, d'où :
   - Si $(f, \vec{a}, b) \in \Delta$:
     $\overline{f}^{(X \oplus \Delta)|_{\mathcal{L}}}(\vec{a}) = \overline{f}^{X \oplus \Delta}(\vec{a}) = b = \overline{f}^{X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}}}(\vec{a})$
   - Si $(f, \vec{a}, b) \notin \Delta$:
     $\overline{f}^{(X \oplus \Delta)|_{\mathcal{L}}}(\vec{a}) = \overline{f}^{X \oplus \Delta}(\vec{a}) = \overline{f}^X(\vec{a}) = \overline{f}^{X|_{\mathcal{L}}}(\vec{a}) = \overline{f}^{X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}}}(\vec{a})$

Donc d'après la définition 1.1.1 p.16 $(X \oplus \Delta)|_{\mathcal{L}}$ et $X|_{\mathcal{L}} \oplus \Delta|_{\mathcal{L}}$ sont les mêmes structures.

□

A.2 Les programmes impératifs

Composition de programmes

Le lemme suivant est vrai pour tout système de transition d'états déterministe, quelle que soit les règles de transition. C'est la raison pour laquelle nous avons adopté la notation générale η pour les états d'un système de transition. Dans le cas des programmes impératifs nous aurons donc $\eta = P \star X$.

Lemme A.2.1. *(Chaînes d'exécution)*
Pour tout $i, j \in \mathbb{N}$:
Si $\eta_1 \succ_i \eta_2$ et $\eta_1 \succ_{i+j} \eta_3$ alors $\eta_2 \succ_j \eta_3$.

Démonstration. Par induction sur i :

$\boldsymbol{i = 0}$

Dans ce cas $\eta_1 = \eta_2$

Donc si $\eta_1 \succ_{0+j} \eta_3$

Alors $\eta_2 \succ_j \eta_3$

$\boldsymbol{i = 1}$

Dans ce cas $\eta_1 \succ \eta_2$ (1)

Comme $\eta_1 \succ_{1+j} \eta_3$ il existe η' tel que :

- $\eta_1 \succ \eta'$ (2)
- $\eta' \succ_j \eta_3$ (3)

Comme le système de transition est déterministe, par (1) et (2) nous avons : $\eta_2 = \eta'$

Donc avec (3) : $\eta_2 \succ_j \eta_3$

$\boldsymbol{i \Rightarrow i + 1}$

Dans ce cas $\eta_1 \succ_{1+i} \eta_2$ (1)

Comme $\eta_1 \succ_{1+i+j} \eta_3$ il existe η' tel que :

- $\eta_1 \succ \eta'$ (2)
- $\eta' \succ_{i+j} \eta_3$ (3)

Comme le cas $j = 1$ est vrai, avec (1) et (2) nous avons : $\eta' \succ_i \eta_2$

Avec (3), nous avons donc d'après l'hypothèse d'induction : $\eta_2 \succ_j \eta_3$

□

Lemme (2.2.7 p.49). *(Commutation transitive de contexte)*
Soit $P_1 \downarrow X$ et $i \leq time(P_1, X)$.
S'il existe P', X' et P_2 tels que $P_1 P_2 \star X \succ_i P' P_2 \star X'$
alors pour tout P_3 : $P_1 P_3 \star X \succ_i P' P_3 \star X'$

La preuve de ce lemme n'utilise pas les règles du système de transition, uniquement son déterminisme et la commutation de contexte :

Démonstration. Par induction sur $time(P_1, X)$:

$\boldsymbol{time(P_1, X) = 0}$

Dans ce cas $i = 0$.

S'il existe P', X' et P_2 tels que $P_1 P_2 \star X \succ_0 P' P_2 \star X'$

Alors : $P_1 P_2 = P' P_2$ et $X = X'$

$P_1P_2 = P'P_2$, donc[1] $P_1 = P'$.
Donc pour tout P_3 : $P_1P_3 = P'P_3$
Or $X = X'$, donc $P_1P_3 \star X \succ_0 P'P_3 \star X'$

$\boldsymbol{time(P_1, X) = t + 1}$

$time(P_1, X) \neq 0$ donc $P_1 \neq \{\}$.
Donc il existe une commande c et un programme P_1' tels que $P_1 = cP_1'$.
D'où, d'après le système de transition 2.2.4, il existe P'' et X'' tels que :

$$cP_1'P_2 \star X \succ P''P_1'P_2 \star X'' \ (1)$$

Le cas $i = 0$ est le même que plus haut, donc supposons $i = j + 1$.
Par hypothèse $j + 1 \leq time(cP_1', X) = t + 1$, donc $j \leq t$.
S'il existe P', X' et P_2 tels que $cP_1'P_2 \star X \succ_{j+1} P'P_2 \star X'$
Alors, d'après le lemme A.2.1 et (1) :

$$P''P_1'P_2 \star X'' \succ_j P'P_2 \star X' \ (2)$$

Par hypothèse $cP_1' \downarrow X$ donc : $cP_1' \star X \succ_{time(cP_1',X)} \{\} \star cP_1'(X)$
Or, par commutation de contexte sur (1) :
$cP_1' \star X \succ P''P_1' \star X''$
Donc, parce que $time(cP_1', X) \geq 1$, nous avons d'après le lemme A.2.1 :
$P''P_1' \star X'' \succ_{time(cP_1',X)-1} \{\} \star cP_1'(X)$
Donc $P''P_1' \downarrow X''$ et $time(P''P_1', X'') = time(cP_1', X) - 1 = t$.

$P''P_1' \downarrow X''$ et $j \leq t = time(P''P_1', X'')$
Donc d'après l'hypothèse d'induction sur (2), pour tout P_3 :
$P''P_1'P_3 \star X'' \succ_j P'P_3 \star X'$
Or, par commutation de contexte sur (1), pour tout P_3 :
$cP_1'P_3 \star X \succ P''P_1'P_3 \star X''$
Donc pour tout P_3 : $cP_1'P_3 \star X \succ_{j+1} P'P_3 \star X'$
D'où $P_1P_3 \star X \succ_i P'P_3 \star X'$

□

Lemme A.2.2. *(Terminaison des préfixes)*
SI $P_1P_2 \downarrow X$ alors $P_1 \downarrow X$

Là encore la preuve de ce lemme n'utilise pas les règles du système de transition, uniquement son déterminisme et la commutation de contexte :

Démonstration. Par induction sur $time(P_1P_2, X)$:

$\boldsymbol{time(P_1P_2, X) = 0}$

Dans ce cas $P_1P_2 \star X = \{\} \star P_1P_2(X)$
Donc $P_1P_2 = \{\}$, d'où $P_1 = \{\}$
Et $P_1 \downarrow X$

1. ↑ La preuve se fait par induction sur P_1.

$\boldsymbol{time(P_1P_2, X) = t + 1}$

Si $P_1 = \{\}$ alors $P_1 \downarrow X$

Sinon il existe c et P_1' tels que $P_1 = cP_1'$

Dans ce cas, d'après le système de transition 2.2.4, il existe P' et X' tels que :

$$cP_1' \star X \succ P'P_1' \star X' \ (1)$$

D'où, par commutation de contexte :

$cP_1'P_2 \star X \succ P'P_1'P_2 \star X'$

Or par hypothèse $cP_1'P_2 \downarrow X$ et $time(cP_1'P_2, X) = t + 1$ donc :

$cP_1'P_2 \star X \succ_{t+1} \{\} \star cP_1'P_2(X)$

D'où, d'après le lemme A.2.1 :

$P'P_1'P_2 \star X' \succ_t \{\} \star cP_1'P_2(X)$

Donc $P'P_1'P_2 \downarrow X'$, avec $time(P'P_1'P_2, X') = t$.

Or, d'après l'hypothèse d'induction sur t : $P'P_1' \downarrow X'$

Donc par définition $P'P_1' \star X' \succ_{time(P'P_1', X)} \{\} \star P'P_1'(X')$

Or (1) : $cP_1' \star X \succ P'P_1' \star X'$

Donc $cP_1' \star X \succ_{time(P'P_1', X)+1} \{\} \star P'P_1'(X')$

Et $P_1 \downarrow X$

□

Proposition (**2.2.9** p.49)**.** *(Composition de programmes)*
$P_1P_2 \downarrow X$ si et seulement si $P_1 \downarrow X$ et $P_2 \downarrow P_1(X)$, tel que :

1. $P_1P_2(X) = P_2(P_1(X))$
2. $time(P_1P_2, X) = time(P_1, X) + time(P_2, P_1(X))$

Comme elle n'est conséquence que des lemmes précédents, la preuve de la proposition n'utilise pas les règles du système de transition, uniquement son déterminisme et la commutation de contexte :

Démonstration. La preuve est faite par double implication :

$\Rightarrow$ Supposons que $P_1P_2 \downarrow X$.

$P_1P_2 \downarrow X$ donc par définition :

$P_1P_2 \star X \succ_{time(P_1P_2, X)} \{\} \star P_1P_2(X)$

Comme $P_1P_2 \downarrow X$, d'après le lemme A.2.2 : $P_1 \downarrow X$

Donc, d'après le corollaire 2.2.8 :

$P_1P_2 \star X \succ_{time(P_1, X)} P_2 \star P_1(X)$

Nous avons $time(P_1P_2, X) \geq time(P_1, X)$, car :

- Si $P_2 = \{\}$ alors $P_1P_2 = P_1$
  Donc $time(P_1P_2, X) = time(P_1, X)$
- Sinon $P_2 \star P_1(X)$ n'est pas un état terminal
  Donc $time(P_1P_2, X) > time(P_1, X)$

Donc d'après le lemme A.2.1 :

$P_2 \star P_1(X) \succ_{time(P_1P_2, X) - time(P_1, X)} \{\} \star P_1P_2(X)$

D'où $P_2 \downarrow P_1(X)$

avec $P_2(P_1(X)) = P_1P_2(X)$

et $time(P_2, P_1(X)) = time(P_1P_2, X) - time(P_1, X)$

$\Leftarrow$ Supposons que $P_1 \downarrow X$ et $P_2 \downarrow P_1(X)$.
$P_1 \downarrow X$ donc d'après le corollaire 2.2.8 :
$P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$
$P_2 \downarrow P_1(X)$ donc par définition :
$P_2 \star P_1(X) \succ_{time(P_2,P_1(X))} \{\} \star P_2(P_1(X))$
D'où par transitivité :
$P_1P_2 \star X \succ_{time(P_1,X)+time(P_2,P_1(X))} \{\} \star P_2(P_1(X))$
Donc $P_1P_2 \downarrow X$
avec $P_1P_2(X) = P_2(P_1(X))$
et $time(P_1P_2, X) = time(P_1, X) + time(P_2, P_1(X))$

□

Mises à jour d'un programme impératif

Lemme A.2.3. *(Finitude des mises à jour)*
Si $P \downarrow X$ alors $card(\Delta(P, X)) \leq time(P, X)$.

Démonstration. Si $P \downarrow X$ alors $\forall i \geq time(P, X)\ \tau_P^i(X) = P(X)$.
Donc $\forall i \geq time(P, X)\ \tau_P^{i+1}(X) \ominus \tau_P^i(X) = P(X) - P(X) = \varnothing$.
D'où $\Delta(P, X) = \bigcup_{0 \leq i < time(P,X)} \tau_P^{i+1}(X) \ominus \tau_P^i(X)$.
Or, comme $\tau_P^{i+1}(X) \ominus \tau_P^i(X)$ est vide ou un singleton
Nous avons donc que $card(\tau_P^{i+1}(X) \ominus \tau_P^i(X)) \leq 1$.

$$\begin{aligned}
\text{Donc } card(\Delta(P, X)) &= card(\bigcup_{0 \leq i < time(P,X)} \tau_P^{i+1}(X) \ominus \tau_P^i(X)) \\
&\leq \sum_{0 \leq i < time(P,X)} card(\tau_P^{i+1}(X) \ominus \tau_P^i(X)) \\
&\leq \sum_{0 \leq i < time(P,X)} 1 \\
&= time(P, X)
\end{aligned}$$

□

Lemme A.2.4. *Pour toute structure X et Y du même langage*
Si $u \in Y \ominus X$ alors $(Y \ominus (X \oplus \{u\})) \cup \{u\} = Y \ominus X$

Démonstration. Soit $u = (\varphi, \vec{\alpha}, \beta)$.
Comme $u \in Y \ominus X$, nous avons $\overline{\varphi}^Y(\vec{\alpha}) = \beta$ et $\overline{\varphi}^Y(\vec{\alpha}) \neq \overline{\varphi}^X(\vec{\alpha})$.

Si $f \neq \varphi$ alors $\overline{f}^{X \oplus \{u\}}(\vec{a}) = \overline{f}^X(\vec{a})$
Sinon, si $\vec{a} \neq \vec{\alpha}$ alors $\overline{\varphi}^{X \oplus \{u\}}(\vec{a}) = \overline{\varphi}^X(\vec{a})$
Sinon $\overline{\varphi}^{X \oplus \{u\}}(\vec{\alpha}) = \beta$, d'où :

$$\begin{aligned}
Y \ominus (X \oplus \{u\}) &= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \mid \overline{f}^Y(\vec{a}) \neq \overline{f}^{X \oplus \{u\}}(\vec{a})\} \\
&= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \mid f \neq \varphi, \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \mid \vec{a} \neq \vec{\alpha}, \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \mid \overline{\varphi}^Y(\vec{\alpha}) \neq \beta\} \\
(Y \ominus X) \backslash \{u\} &= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u \mid \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\} \\
&= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u \mid f \neq \varphi, \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \neq u \mid \vec{a} \neq \vec{\alpha}, \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \neq u \mid \overline{\varphi}^Y(\vec{\alpha}) \neq \overline{\varphi}^X(\vec{\alpha})\}
\end{aligned}$$

Comme $f \neq \varphi$ nous avons $(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u$, d'où :
$\{(f, \vec{a}, \overline{f}^Y(\vec{a})) \mid f \neq \varphi, \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\}$
$= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u \mid f \neq \varphi, \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\}$
Comme $\vec{a} \neq \vec{\alpha}$ nous avons $(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u$, d'où :
$\{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \mid \vec{a} \neq \vec{\alpha}, \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\}$
$= \{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \neq u \mid \vec{a} \neq \vec{\alpha}, \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\}$
Comme $\overline{\varphi}^Y(\vec{\alpha}) = \beta$ nous avons que
$\{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \mid \overline{\varphi}^Y(\vec{\alpha}) \neq \beta\}$
$= \varnothing$
Comme $(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) = u$ nous avons que
$\{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \neq u \mid \overline{\varphi}^Y(\vec{\alpha}) \neq \overline{\varphi}^X(\vec{\alpha})\}$
$= \varnothing$
Donc $Y \ominus (X \oplus \{u\}) = (Y \ominus X) \backslash \{u\}$.

$$\begin{aligned} \text{D'où } (Y \ominus (X \oplus \{u\})) \cup \{u\} &= ((Y \ominus X) \backslash \{u\}) \cup \{u\} \\ &= Y \ominus X \text{ car } u \in Y \ominus X \end{aligned}$$

□

Proposition (**2.2.12** p.52)**.** *(Mises à jour sans écrasement)*
Si $P \downarrow X$ sans écrasement alors $\Delta(P, X) = P(X) \ominus X$.

Démonstration. Par induction sur $time(P, X)$:

$\boldsymbol{time(P, X) = 0}$

Dans ce cas $P \star X = \{\} \star P(X)$, donc :

$$\begin{aligned} \Delta(P, X) &= \bigcup_{0 \leq i < time(P,X)} \tau_P^{i+1}(X) \ominus \tau_P^i(X) \\ &= \varnothing \\ &= P(X) \ominus X \end{aligned}$$

$\boldsymbol{time(P, X) = t + 1}$

$time(P, X) \neq 0$ donc il existe une commande c et un programme impératif $\tilde{P}$ tels que $P = c\tilde{P}$

Par le système de transition 2.2.4 il existe P' et X' tels que :
$c\tilde{P} \star X \succ P'\tilde{P} \star X'$
Or par hypothèse :
$c\tilde{P} \star X \succ_{t+1} \{\} \star c\tilde{P}(X)$
Donc d'après le lemme A.2.1 :
$P'\tilde{P} \star X' \succ_t \{\} \star c\tilde{P}(X)$
Donc $P'\tilde{P} \downarrow X'$
avec $time(P'\tilde{P}, X') = t$ et $P'\tilde{P}(X') = c\tilde{P}(X)$

Pour tout $0 \leq i \leq time(P'\tilde{P}, X')$,
Prouvons que $\tau_{c\tilde{P}}^{i+1}(X) = \tau_{P'\tilde{P}}^i(X')$:
Par définition $P'\tilde{P} \star X' \succ_i \tau_{X'}^i(P'\tilde{P}) \star \tau_{P'\tilde{P}}^i(X')$
Or $c\tilde{P} \star X \succ P'\tilde{P} \star X'$
Donc par transitivité $c\tilde{P} \star X \succ_{i+1} \tau_{X'}^i(P'\tilde{P}) \star \tau_{P'\tilde{P}}^i(X')$
Or par définition $c\tilde{P} \star X \succ_{i+1} \tau_X^{i+1}(c\tilde{P}) \star \tau_{c\tilde{P}}^{i+1}(X)$

Donc, comme le système de transition est déterministe :
$\tau^{i+1}_{c\tilde{P}}(X) = \tau^{i}_{P'\tilde{P}}(X')$

$$\begin{aligned}
\Delta(c\tilde{P}, X) &= \bigcup_{0 \leq i < time(c\tilde{P},X)} \tau^{i+1}_{c\tilde{P}}(X) \ominus \tau^{i}_{c\tilde{P}}(X) \\
&= (\tau^{1}_{c\tilde{P}}(X) \ominus \tau^{0}_{c\tilde{P}}(X)) \cup \bigcup_{0 \leq i < time(c\tilde{P},X)-1} \tau^{i+2}_{c\tilde{P}}(X) \ominus \tau^{i+1}_{c\tilde{P}}(X) \\
&= (X' \ominus X) \cup \bigcup_{0 \leq i < time(P'\tilde{P},X')} \tau^{i+1}_{P'\tilde{P}}(X') \ominus \tau^{i}_{P'\tilde{P}}(X') \\
&= (X' \ominus X) \cup \Delta(P'\tilde{P}, X')
\end{aligned}$$

Comme $\Delta(c\tilde{P}, X)$ est consistant, $\Delta(P'\tilde{P}, X')$ l'est aussi.
Donc $P'\tilde{P}$ est sans écrasement sur X'.

$P'\tilde{P} \downarrow X'$ sans écrasement, et $time(P'\tilde{P}, X') = t$
Donc d'après l'hypothèse d'induction sur t : $\Delta(P'\tilde{P}, X') = P'\tilde{P}(X') \ominus X'$
Or $P'\tilde{P}(X') = c\tilde{P}(X)$, donc :

$$\begin{aligned}
\Delta(c\tilde{P}, X) &= (X' \ominus X) \cup \Delta(P'\tilde{P}, X') \\
&= (X' \ominus X) \cup (P'\tilde{P}(X') \ominus X') \\
&= (X' \ominus X) \cup (c\tilde{P}(X) \ominus X')
\end{aligned}$$

Comme $X' = \tau^{1}_{c\tilde{P}}(X)$, il y a deux cas :

$\boldsymbol{X' \ominus X = \varnothing}$

Dans ce cas $X' = X$, donc :

$$\begin{aligned}
\Delta(c\tilde{P}, X) &= \varnothing \cup (c\tilde{P}(X) \ominus X') \\
&= c\tilde{P}(X) \ominus X
\end{aligned}$$

$\boldsymbol{X' \ominus X = \{u\}}$

Dans ce cas $X' = X \oplus \{u\}$, donc :

$$\begin{aligned}
\Delta(c\tilde{P}, X) &= (X' \ominus X) \cup (c\tilde{P}(X) \ominus X') \\
&= \{u\} \cup (c\tilde{P}(X) \ominus (X \oplus \{u\}))
\end{aligned}$$

Prouvons à présent que $u \in c\tilde{P}(X) \ominus X$:
Soit $u = (f, \vec{a}, b)$.
$\{u\} = X' \ominus X = \tau^{1}_{c\tilde{P}}(X) \ominus \tau^{0}_{c\tilde{P}}(X)$, donc $u \in \Delta(c\tilde{P}, X)$.
Par hypothèse $\Delta(c\tilde{P}, X)$ est consistant donc il n'existe dans $\Delta(c\tilde{P}, X)$ pas d'autre mise à jour à cet emplacement.
D'où pour tout $1 \leq i \leq time(c\tilde{P}, X)$: $\overline{f}^{\tau^{i}_{c\tilde{P}}(X)}(\vec{a}) = b$.
En particulier $\tau^{time(c\tilde{P},X)}_{c\tilde{P}}(X) = c\tilde{P}(X)$, donc $\overline{f}^{c\tilde{P}(X)}(\vec{a}) = b$.
Or $\overline{f}^{X}(\vec{a}) \neq b$ car $X' \ominus X = \{u\}$.
Donc $u = (f, \vec{a}, b) \in c\tilde{P}(X) \ominus X$.

D'où d'après le lemme A.2.4 : $\Delta(c\tilde{P}, X) = c\tilde{P}(X) \ominus X$

Dans tous les cas : $\Delta(P, X) = P(X) \ominus X$

□

A.3 Graphes d'exécution

Finitude d'un graphe d'exécution

Définition A.3.1. (Longueur d'un programme)

$$\begin{aligned} \mathit{length}(\{\}) &=_{def} 0 \\ \mathit{length}(\{c; s\}) &=_{def} \mathit{length}(c) + \mathit{length}(\{s\}) \end{aligned}$$

$$\begin{aligned} \mathit{length}(ft_1 \ldots t_k := t_0) &=_{def} 1 \\ \mathit{length}(\texttt{if } F\ P_1 \texttt{ else } P_2) &=_{def} 1 + \mathit{length}(P_1) + \mathit{length}(P_2) \\ \mathit{length}(\texttt{while } F\ P_1) &=_{def} 1 + \mathit{length}(P_1) \\ \mathit{length}(\texttt{loop } n \texttt{ except } F\ P_1) &=_{def} 1 + \mathit{length}(P_1) \end{aligned}$$

Lemme (**3.1.5** p.63)**.** *(Finitude d'un graphe d'exécution)*

$$\mathit{card}(\mathcal{G}(P)) \leq \mathit{length}(P) + 1$$

Démonstration. Par induction sur P :

$\boldsymbol{P = \{\}}$

Dans ce cas :

$\mathcal{G}(P) = \boldsymbol{\{}\{\}\boldsymbol{\}}$

Donc :

$$\mathit{card}(\mathcal{G}(P)) = 1 = \mathit{length}(P) + 1$$

$\boldsymbol{P = \{u; s\}}$

Dans ce cas :

$\mathcal{G}(P) = \boldsymbol{\{}\{u; s\}\boldsymbol{\}} \cup \mathcal{G}(\{s\})$

Donc :

$$\begin{aligned} \mathit{card}(\mathcal{G}(P)) &\leq 1 + \mathit{card}(\mathcal{G}(\{s\})) \\ &\leq 1 + \mathit{length}(\{s\}) + 1 \text{ (hypothèse d'induction)} \\ &= \mathit{length}(P) + 1 \end{aligned}$$

$\boldsymbol{P = \{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\}}$

Dans ce cas :

$$\begin{aligned} \mathcal{G}(P) &= \boldsymbol{\{}\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\}\boldsymbol{\}} \\ &\quad \cup \mathcal{G}(\{s_1\})\{s\} \cup \mathcal{G}(\{s_2\})\{s\} \cup \mathcal{G}(\{s\}) \\ &= \boldsymbol{\{}\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\}\boldsymbol{\}} \\ &\quad \cup \mathcal{G}(\{s_1\})\{s\} \backslash \boldsymbol{\{}\{s\}\boldsymbol{\}} \cup \mathcal{G}(\{s_2\})\{s\} \backslash \boldsymbol{\{}\{s\}\boldsymbol{\}} \cup \mathcal{G}(\{s\}) \end{aligned}$$

Car $P' \in \mathcal{G}(\{s\})$, et $\{\} \in \mathcal{G}(P_1), \mathcal{G}(P_2)$

implique que $P' \in \mathcal{G}(\{s_1\})\{s\}, \mathcal{G}(\{s_2\})\{s\}$

Donc :

$$\begin{aligned} \mathit{card}(\mathcal{G}(P)) &\leq 1 + (\mathit{card}(\mathcal{G}(\{s_1\})) - 1) + (\mathit{card}(\mathcal{G}(\{s_2\})) - 1) \\ &\quad + \mathit{card}(\mathcal{G}(\{s\})) \\ &\leq 1 + \mathit{length}(\{s_1\}) + \mathit{length}(\{s_2\}) + \mathit{length}(\{s\}) + 1 \\ &\quad \text{(hypothèse d'induction)} \\ &= \mathit{length}(P) + 1 \end{aligned}$$

$\boldsymbol{P = \{\texttt{while}\ F\ \{s_1\}; s\}}$

Dans ce cas :

$\mathcal{G}(P) = \mathcal{G}(P_1)\{\texttt{while}\ F\ \{s_1\}; s\} \cup \mathcal{G}(\{s\})$

Donc :

$$\begin{aligned} card(\mathcal{G}(P)) &\leq card(\mathcal{G}(\{s_1\})) + card(\mathcal{G}(\{s\})) \\ &\leq 1 + length(\{s_1\}) + length(\{s\}) + 1 \\ &\quad \text{(hypothèse d'induction)} \\ &= length(P) + 1 \end{aligned}$$

$\boldsymbol{P = \{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s\}}$

De même.

□

Composition de graphes d'exécution

Lemme (3.1.6p.63). *(Composition de graphes d'exécution)*

$$\mathcal{G}(P_1P_2) = \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$$

Démonstration. Par induction on P_1 :

$\boldsymbol{P_1 = \{\}}$

Dans ce cas :

$$\begin{aligned} \mathcal{G}(P_1P_2) &= \mathcal{G}(P_2) \\ &= \boldsymbol{\{}P_2\boldsymbol{\}} \cup \mathcal{G}(P_2)\ (\text{car } P_2 \in \mathcal{G}(P_2)) \\ &= \boldsymbol{\{}\{\}\boldsymbol{\}}P_2 \cup \mathcal{G}(P_2) \\ &= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2) \end{aligned}$$

$\boldsymbol{P_1 = \{u; s\}}$

Dans ce cas :

$$\begin{aligned} \mathcal{G}(P_1P_2) &= \mathcal{G}(\{u; s; s_{P_2}\}) \\ &= \boldsymbol{\{}\{u; s; s_{P_2}\}\boldsymbol{\}} \cup \mathcal{G}(\{s; s_{P_2}\}) \\ &= \boldsymbol{\{}\{u; s\}\boldsymbol{\}}P_2 \cup \mathcal{G}(\{s\})P_2 \cup \mathcal{G}(P_2)\ \text{(hypothèse d'induction)} \\ &= (\ \boldsymbol{\{}\{u; s\}\boldsymbol{\}} \cup \mathcal{G}(\{s\})\ )P_2 \cup \mathcal{G}(P_2) \\ &= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2) \end{aligned}$$

$\boldsymbol{P_1 = \{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}}$

Dans ce cas :

$$\begin{aligned} \mathcal{G}(P_1P_2) &= \boldsymbol{\{}P_1P_2\boldsymbol{\}} \cup\ \mathcal{G}(\{s_1\})\{s; s_{P_2}\} \cup \mathcal{G}(\{s_2\})\{s; s_{P_2}\} \cup \mathcal{G}(\{s; s_{P_2}\}) \\ &= \boldsymbol{\{}P_1\boldsymbol{\}}P_2 \cup\ \mathcal{G}(\{s_1\})\{s\}P_2 \cup \mathcal{G}(\{s_2\})\{s\}P_2 \\ &\quad \cup\ \mathcal{G}(\{s\})P_2 \cup \mathcal{G}(P_2)\ \text{(hypothèse d'induction)} \\ &= (\ \boldsymbol{\{}P_1\boldsymbol{\}} \cup \mathcal{G}(\{s_1\})\{s\} \cup \mathcal{G}(\{s_2\})\{s\} \cup \mathcal{G}(\{s\})\ )P_2 \cup \mathcal{G}(P_2) \\ &= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2) \end{aligned}$$

$\boldsymbol{P_1 = \{\texttt{while}\ F\ \{s_1\}; s\}}$

Dans ce cas :

$$\begin{aligned} \mathcal{G}(P_1P_2) &= \mathcal{G}(\{\texttt{while}\ F\ \{s_1\}; s; s_{P_2}\}) \\ &= \mathcal{G}(\{s_1\})\{\texttt{while}\ F\ \{s_1\}; s; s_{P_2}\} \cup \mathcal{G}(\{s; s_{P_2}\}) \\ &= \mathcal{G}(\{s_1\})\{\texttt{while}\ F\ \{s_1\}; s\}P_2 \cup \mathcal{G}(\{s\})P_2 \cup \mathcal{G}(P_2) \\ &\quad \text{(hypothèse d'induction)} \\ &= (\ \mathcal{G}(\{s_1\})\{\texttt{while}\ F\ \{s_1\}; s\} \cup \mathcal{G}(\{s\})\ )P_2 \cup \mathcal{G}(P_2) \\ &= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2) \end{aligned}$$

$\boldsymbol{P_1 = \{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s\}}$

De même.

□

Sous-graphes d'exécution

Lemme (**3.1.7** p.63)**.** *(Sous-graphes d'exécution)*
Si $Q \in \mathcal{G}(P)$ alors $\mathcal{G}(Q) \subseteq \mathcal{G}(P)$

Démonstration. Par induction sur P :

$\boldsymbol{P = \{\}}$

Dans ce cas :

$Q \in \mathcal{G}(P) = \boldsymbol{\{}\{\}\boldsymbol{\}}$

Donc $Q = \{\} = P$ et $\mathcal{G}(Q) = \mathcal{G}(P)$

$\boldsymbol{P = \{u; s\}}$

Dans ce cas :

$Q \in \mathcal{G}(P) = \boldsymbol{\{}\{u; s\}\boldsymbol{\}} \cup \mathcal{G}(\{s\})$

Donc $Q = \{u; s\} = P$ ou $Q \in \mathcal{G}(\{s\})$

$\boldsymbol{Q = P}$

Dans ce cas :

$\mathcal{G}(Q) = \mathcal{G}(P)$

$\boldsymbol{Q \in \mathcal{G}(\{s\})}$

Dans ce cas :

D'après l'hypothèse d'induction $\mathcal{G}(Q) \subseteq \mathcal{G}(\{s\})$

Donc $\mathcal{G}(Q) \subseteq \mathcal{G}(\{s\}) \subseteq \boldsymbol{\{}\{u; s\}\boldsymbol{\}} \cup \mathcal{G}(\{s\}) = \mathcal{G}(P)$

$\boldsymbol{P = \{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}}$

Dans ce cas :

$Q \in \mathcal{G}(P)$

$= \boldsymbol{\{}\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}\boldsymbol{\}} \cup \mathcal{G}(\{s_1\})\{s\} \cup \mathcal{G}(\{s_2\})\{s\} \cup \mathcal{G}(\{s\})$

Donc $Q = \{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\} = P$

ou $Q \in \mathcal{G}(\{s_1\})\{s\}$ ou $Q \in \mathcal{G}(\{s_2\})\{s\}$ ou $Q \in \mathcal{G}(\{s\})$

$\boldsymbol{Q = P}$

Dans ce cas :

$\mathcal{G}(Q) = \mathcal{G}(P)$

$\boldsymbol{Q \in \mathcal{G}(\{s_1\})\{s\}}$

Dans ce cas :

Il existe $Q' \in \mathcal{G}(\{s_1\})$ tel que $Q = Q'\{s\}$

D'après l'hypothèse d'induction $\mathcal{G}(Q') \subseteq \mathcal{G}(\{s_1\})$

Donc $\mathcal{G}(Q')\{s\} \subseteq \mathcal{G}(\{s_1\})\{s\}$

D'après le lemme 3.1.6 $\mathcal{G}(Q'\{s\}) = \mathcal{G}(Q')\{s\} \cup \mathcal{G}(\{s\})$

Donc :

$$\begin{aligned}
\mathcal{G}(Q) &= \mathcal{G}(Q')\{s\} \cup \mathcal{G}(\{s\}) \\
&\subseteq \mathcal{G}(\{s_1\})\{s\} \cup \mathcal{G}(\{s\}) \\
&\subseteq \boldsymbol{\{}\{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}\boldsymbol{\}} \\
&\quad \cup \mathcal{G}(\{s_1\})\{s\} \cup \mathcal{G}(\{s_2\})\{s\} \cup \mathcal{G}(\{s\}) \\
&= \mathcal{G}(P)
\end{aligned}$$

$\boldsymbol{Q \in \mathcal{G}(\{s_2\})\{s\}}$

De même.

$\boldsymbol{Q \in \mathcal{G}(\{s\})}$

Dans ce cas :

D'après l'hypothèse d'induction $\mathcal{G}(Q) \subseteq \mathcal{G}(\{s\})$

Donc :

$$\begin{aligned}\mathcal{G}(Q) &\subseteq \mathcal{G}(\{s\})\\ &\subseteq \{\{\texttt{if } F \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}; s\}\}\\ &\quad \cup \mathcal{G}(\{s_1\})\{s\} \cup \mathcal{G}(\{s_2\})\{s\} \cup \mathcal{G}(\{s\})\\ &= \mathcal{G}(P)\end{aligned}$$

$\boldsymbol{P = \{\texttt{while } F\ \{s_1\}; s\}}$

Dans ce cas :

$Q \in \mathcal{G}(P)$

$= \mathcal{G}(\{s_1\})\{\texttt{while } F\ \{s_1\}; s\} \cup \mathcal{G}(\{s\})$

$= \mathcal{G}(\{s_1\})P \cup \mathcal{G}(\{s\})$

Donc $Q \in \mathcal{G}(\{s_1\})P$ ou $Q \in \mathcal{G}(\{s\})$

$\boldsymbol{Q \in \mathcal{G}(\{s_1\})P}$

Dans ce cas :

Il existe $Q' \in \mathcal{G}(\{s_1\})$ tel que $Q = Q'P$

D'après l'hypothèse d'induction $\mathcal{G}(Q') \subseteq \mathcal{G}(\{s_1\})$

Donc $\mathcal{G}(Q')P \subseteq \mathcal{G}(\{s_1\})P$

D'après le lemme 3.1.6 $\mathcal{G}(Q'P) = \mathcal{G}(Q')P \cup \mathcal{G}(P)$

Donc :

$$\begin{aligned}\mathcal{G}(Q) &= \mathcal{G}(Q')P \cup \mathcal{G}(P)\\ &\subseteq \mathcal{G}(\{s_1\})P \cup \mathcal{G}(P)\\ &= \mathcal{G}(\{s_1\})P \cup \mathcal{G}(\{s_1\})P \cup \mathcal{G}(\{s\})\\ &= \mathcal{G}(\{s_1\})P \cup \mathcal{G}(\{s\})\\ &= \mathcal{G}(P)\end{aligned}$$

$\boldsymbol{Q \in \mathcal{G}(\{s\})}$

Dans ce cas :

D'après l'hypothèse d'induction $\mathcal{G}(Q) \subseteq \mathcal{G}(\{s\})$

Donc :

$$\begin{aligned}\mathcal{G}(Q) &\subseteq \mathcal{G}(\{s\})\\ &\subseteq \mathcal{G}(\{s_1\})P \cup \mathcal{G}(\{s\})\\ &= \mathcal{G}(P)\end{aligned}$$

$\boldsymbol{P = \{\texttt{loop } n \texttt{ except } F\ \{s_1\}; s\}}$

De même.

□

A.4 Traduction des ASMs

Traduction d'un programme en forme normale

Rappel. La traduction P_{Π} d'un pas d'ASM est donnée à la table A.1 p.131.

TABLE A.1 – Simulation d'un pas de Π

$$
\begin{array}{l}
P_\Pi = \{ \\
\quad v_{t_1} := t_1 \ ; \\
\quad v_{t_2} := t_2 \ ; \\
\quad \vdots \\
\quad v_{t_r} := t_r \ ; \\
\quad \texttt{if } v_{F_1} \texttt{ then } \{ \\
\qquad f_1^1(\vec{v}_{t_1^1}) := v_{t_1^1} \ ; \\
\qquad f_2^1(\vec{v}_{t_2^1}) := v_{t_2^1} \ ; \\
\qquad \vdots \\
\qquad f_{m_1}^1(\vec{v}_{t_{m_1}^1}) := v_{t_{m_1}^1} \ ; \\
\qquad \texttt{skip } (c+m) - (1+m_1) \\
\quad \} \texttt{ else } \{ \\
\quad \texttt{if } v_{F_2} \texttt{ then } \{ \\
\qquad f_1^2(\vec{v}_{t_1^2}) := v_{t_1^2} \ ; \\
\qquad f_2^2(\vec{v}_{t_2^2}) := v_{t_2^2} \ ; \\
\qquad \vdots \\
\qquad f_{m_2}^2(\vec{v}_{t_{m_2}^2}) := v_{t_{m_2}^2} \ ; \\
\qquad \texttt{skip } (c+m) - (2+m_2) \\
\quad \} \texttt{ else } \{ \\
\quad \vdots \\
\quad \texttt{if } v_{F_c} \texttt{ then } \{ \\
\qquad f_1^c(\vec{v}_{t_1^c}) := v_{t_1^c} \ ; \\
\qquad f_2^c(\vec{v}_{t_2^c}) := v_{t_2^c} \ ; \\
\qquad \vdots \\
\qquad f_{m_c}^c(\vec{v}_{t_{m_c}^c}) := v_{t_{m_c}^c} \ ; \\
\qquad \texttt{skip } (c+m) - (c+m_c) \\
\quad \} \ ; \ \ldots \ \} \ ; \\
\}
\end{array}
$$

Proposition (**4.1.5** p.89)**.** *(Traduction d'un pas d'ASM)*

1. $(P_\Pi(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$
2. $time(P_\Pi, X) = r + c + m$

où :

- *r est le nombre de termes lus par Π*
- *c est le nombre d'états reconnus par Π*
- *m est le degré de parallélisme de Π*

Démonstration. Les variables temporaires $\vec{v_t}$ n'apparaissent pas parmi les termes $\vec{t} = Read(\Pi)$, donc après l'initialisation $\vec{v_t} := \vec{t}$ l'état est $X' = X \oplus \{(\vec{v_t}, \vec{t}^X)\}$, et pour tout $k \in \{1, \dots r\}$: $\overline{v_{t_k}}^{X'} = \overline{t_k}^X$.

Le programme restant est $\Pi[\vec{v}/\vec{t}]^{tr}$ avec les commandes `skip` ajoutées à la fin de chaque bloc.

Les symboles mis à jour sont les mêmes que ceux de Π, donc comme les $\vec{v_t}$ sont fraîches elles ne sont pas mises à jour dans la suite du programme. Donc pour tous les états suivants Y nous avons que : $\overline{v_{t_k}}^Y = \overline{v_{t_k}}^{X'}$

Donc pour tout $k \in \{1, \dots r\}$ et pour tous les états suivants Y : $\overline{v_{t_k}}^Y = \overline{t_k}^X$

Le programme restant est une imbrication de conditionnelles où les F_i sont des gardes : pour tout état Y une et une seule F_i est vraie.

Soit F_j la formule vraie dans X.

Or nous avons prouvé que pour tous les états suivants Y et toutes les conditionnelles F_i : $\overline{v_{F_i}}^Y = \overline{F_i}^X$.

Donc durant les états suivants seule v_{F_j} est vraie, donc seul le bloc de `if` v_{F_j} est exécuté.

Les commandes de ce bloc sont : $f_1^j(\vec{v_{t_1^j}}) := v_{t_1^j}; \dots; f_{m_j}^j(\vec{v_{t_{m_j}^j}}) := v_{t_{m_j}^j}$; suivies de $(c + m) - (j + m_j)$ commandes `skip`.

Comme pour tous les états suivants Y : $\overline{v_{t_k}}^Y = \overline{t_k}^X$, l'ensemble de mises à jours induit par ce bloc est :

$$\{(f_1^j, \vec{t_1^j}^X, \overline{t_1^j}^X) \dots (f_{m_j}^j, \vec{t_{m_j}^j}^X, \overline{t_{m_j}^j}^X)\} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$$

Donc : $\Delta(P_\Pi, X) = \{(v_t, \bar{t}^X) \mid t \in Read(\Pi)\} \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$

$P_\Pi \in$ `LoopC` donc d'après le corollaire 2.2.10 il est terminal.

Comme Π est sous forme normale : $\Delta(\Pi, X|_{\mathcal{L}(\Pi)}) = \tau_\Pi(X|_{\mathcal{L}(\Pi)}) \ominus X|_{\mathcal{L}(\Pi)}$

D'où $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$ est consistant.

Or les variables temporaires $\vec{v_t}$ ne sont mises à jour qu'une seule fois durant l'initialisation.

Donc $\Delta(P_\Pi, X)$ est consistant également.

Donc d'après la proposition 2.2.12 : $\Delta(P_\Pi, X) = P_\Pi(X) \ominus X$, d'où :

$(P_\Pi(X) \ominus X)|_{\mathcal{L}(\Pi)}$
$= \Delta(P_\Pi, X)|_{\mathcal{L}(\Pi)}$
$= (\{(v_t, \bar{t}^X) \mid t \in Read(\Pi)\} \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)}))|_{\mathcal{L}(\Pi)}$
$= \{(v_t, \bar{t}^X) \mid t \in Read(\Pi)\}|_{\mathcal{L}(\Pi)} \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)})|_{\mathcal{L}(\Pi)}$
$= \varnothing \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$
$= \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$

Enfin, nous avons pour le temps d'exécution :

- L'initialisation des variables temporaires requiert r étapes.
- Il faut j étapes pour arriver au bloc de F_j

– Les mises à jour de ce bloc requièrent m_j étapes.
– Enfin il y a $(c+m)-(j+m_j)$ commandes `skip`.

Donc nous avons :

$$\begin{aligned} time(P_\Pi, X) &= r + (j+m_j) + (c+m) - (j+m_j) \\ &= r + c + m \end{aligned}$$

qui ne dépendent que de Π. □

Condition d'arrêt

Lemme (4.2.1 p.94). *(μ-formule)*

$$time(\Pi, X|_{\mathcal{L}(\Pi)}) = min\{i \in \mathbb{N} \mid \overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true\}$$

$$où\ F_\Pi =_{def} \bigwedge_{t \in Read(\Pi)} v_t = t$$

Démonstration. Comme :

$$time(\Pi, X|_{\mathcal{L}(\Pi)}) = min\{i \in \mathbb{N} \mid \tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})\}$$

Nous devons montrons que :

$$\tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}) \Leftrightarrow \overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true$$

Tout d'abord remarquons que
$\overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true$
$\Leftrightarrow \forall t \in Read(\Pi)\ \overline{v_t}^{P_\Pi^{i+1}(X)} = \overline{t}^{P_\Pi^{i+1}(X)}$ par définition de F_Π
$\Leftrightarrow \forall t \in Read(\Pi)\ \overline{t}^{P_\Pi^{i}(X)} = \overline{t}^{P_\Pi^{i+1}(X)}$ d'après la remarque p.87
$\Leftrightarrow \forall t \in Read(\Pi)\ \overline{t}^{P_\Pi^{i}(X)|_{\mathcal{L}(\Pi)}} = \overline{t}^{P_\Pi^{i+1}(X)|_{\mathcal{L}(\Pi)}}$ car $t \in Read(\Pi)$
$\Leftrightarrow \forall t \in Read(\Pi)\ \overline{t}^{\tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})} = \overline{t}^{\tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})}$ d'après le corollaire 4.1.6 p.91
Il ne reste qu'à finir la double implication :

$\Rightarrow$ Si $\tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})$
Alors $\forall t \in Read(\Pi)\ \overline{t}^{\tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})} = \overline{t}^{\tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})}$
Donc $\overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true$

$\Leftarrow$ Si $\overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true$
Alors $\forall t \in Read(\Pi)\ \overline{t}^{\tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})} = \overline{t}^{\tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})}$
Donc $\tau_\Pi^i(X|_{\mathcal{L}(\Pi)})$ et $\tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})$ coïncide sur $Read(\Pi)$.
Or, d'après le lemme p.37 $\Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}(\Pi)})) = \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}))$, donc :

$$\begin{aligned} \tau_\Pi^{i+2}(X|_{\mathcal{L}(\Pi)}) &= \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}) \oplus \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})) \\ &= (\tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) \oplus \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}(\Pi)}))) \oplus \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})) \\ &= (\tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) \oplus \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}(\Pi)}))) \oplus \Delta(\Pi, \tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})) \\ &= \tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) \oplus \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}(\Pi)})) \\ &= \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}) \end{aligned}$$

D'où $\tau_\Pi^{i+2}(X|_{\mathcal{L}(\Pi)}) \ominus \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}) = \varnothing$

Or comme Π est sous forme normale :
$\Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})) = \tau_\Pi^{i+2}(X|_{\mathcal{L}(\Pi)}) \ominus \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})$
Donc $\Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)})) = \varnothing$
Or $\Delta(\Pi, \tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})) = \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}))$
Donc $\Delta(\Pi, \tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})) = \varnothing$
D'où $\tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)}) + \Delta(\Pi, \tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})) = \tau_\Pi^{i}(X|_{\mathcal{L}(\Pi)})$

□

Lemme (4.2.4 p.95)**.** *Si F_{end} est fausse alors :*

$$\begin{array}{rr} & P_k^{F_{end}}\ P_{k-1}^{F_{end}} \ldots\ P_1^{F_{end}} \\ \succ \texttt{if } \neg F_{end} \texttt{ then } & \{s_{Q_\ell}\}^{F_{end}}\ Q_{\ell-1}^{F_{end}} \ldots\ Q_1^{F_{end}} \\ \succ & Q_\ell^{F_{end}}\ Q_{\ell-1}^{F_{end}} \ldots\ Q_1^{F_{end}} \end{array}$$

tel que $\max\limits_{1\leq i\leq k}\{nest(P_i)+i\} \geq \max\limits_{1\leq j\leq \ell}\{nest(Q_j)+j\}$

Démonstration. Par cas sur P_k :

$\boldsymbol{P_k = \{ft_1 \ldots t_k := t_0; s\}}$

Dans ce cas :

$$\begin{array}{rl} P_k^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} = & \{ft_1 \ldots t_k := t_0; \\ & \texttt{if } \neg F_{end} \texttt{ then } \{s\}^{F_{end}};\}\ P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \\ \succ & \{\texttt{if } \neg F_{end} \texttt{ then } \{s\}^{F_{end}};\}\ P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \\ \succ & \{s\}^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \end{array}$$

$\boldsymbol{P_k = \{\texttt{while } F\ \{s_1\}; s\}}$

Dans ce cas :

$$\begin{array}{rl} P_k^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} = & \{\texttt{while } (F \wedge \neg F_{end})\ \{\texttt{if } \neg F_{end} \texttt{ then } \{s_1\}^{F_{end}};\}; \\ & \texttt{if } \neg F_{end} \texttt{ then } \{s\}^{F_{end}};\}\ P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \end{array}$$

Deux cas :

F est vraie

Dans ce cas :

$$\begin{array}{rl} P_k^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \succ & \{\texttt{if } \neg F_{end} \texttt{ then } \{s_1\}^{F_{end}};\} \\ & P_k^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \\ \succ & \{s_1\}^{F_{end}} P_k^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \end{array}$$

F est fausse

Dans ce cas :

$$\begin{array}{rl} P_k^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \succ & \{\texttt{if } \neg F_{end} \texttt{ then } \{s\}^{F_{end}};\} \\ & P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \\ \succ & \{s\}^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \end{array}$$

$\boldsymbol{P_k = \{\texttt{loop } n\ \{s_1\}; s\}}$

Dans ce cas :

$$\begin{array}{rl} P_k^{F_{end}} P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} = & \{\texttt{loop } n \texttt{ except } F_{end}\ \{\texttt{if } \neg F_{end} \texttt{ then } \{s_1\}^{F_{end}};\}; \\ & \texttt{if } \neg F_{end} \texttt{ then } \{s\}^{F_{end}};\}\ P_{k-1}^{F_{end}} \ldots P_1^{F_{end}} \end{array}$$

Deux cas :

$\boldsymbol{i < n}$

Dans ce cas :

$$\begin{aligned} P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} &\succ \{\texttt{if } \neg F_{end} \texttt{ then } \{s_1\}^{F_{end}};\} \\ & \quad P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \\ &\succ \{s_1\}^{F_{end}} P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \end{aligned}$$

$\boldsymbol{i = n}$

Dans ce cas :

$$\begin{aligned} P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} &\succ \{\texttt{if } \neg F_{end} \texttt{ then } \{s\}^{F_{end}};\} \\ & \quad P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \\ &\succ \{s\}^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \end{aligned}$$

Pour l'imbrication nous regroupons les cas :

1. Élimination de la commande (cas 1, 2.b et 3.b), deux cas :

   $\boldsymbol{s \neq \epsilon}$

   Dans ce cas :
   $\{c; s\}^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \succ_2 \{s\}^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}}$
   avec $nest(\{c; s\}) = max(nest(\{c;\}), nest(\{s\})) \geq nest(\{s\})$
   d'où $nest(\{c; s\}) + k \geq nest(\{s\}) + k$

   $\boldsymbol{s = \epsilon}$

   Dans ce cas :
   $\{c;\}^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \succ_2 P_{k-1}^{F_{end}} \dots P_1^{F_{end}}$
   avec $max(nest(\{c;\}) + k, max_{1 \leq i \leq k-1}\{nest(P_i) + i\})$
   $\geq max_{1 \leq i \leq k-1}\{nest(P_i) + i\}$

2. Entrée dans une boucle (cas 2.a et 3.a), deux cas :

   $\boldsymbol{s_1 \neq \epsilon}$

   Dans ce cas :
   $P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \succ_2 \{s_1\}^{F_{end}} P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}}$
   avec $nest(P_k) = max(nest(\{s_1\}) + 1, nest(\{s\})) \geq nest(\{s_1\}) + 1$
   d'où $nest(P_k) + k \geq nest(\{s_1\}) + (k + 1)$

   $\boldsymbol{s_1 = \epsilon}$

   Dans ce cas :
   $P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}} \succ_2 P_k^{F_{end}} P_{k-1}^{F_{end}} \dots P_1^{F_{end}}$
   Et le programme est inchangé.

□

Sémantique de l'insertion de programme

Lemme (**4.3.2** p.98)**.** *Si $P \neq \{\}$ et Q est terminal alors*

$$P[Q] \star X \succ_{1+time(Q, \tau_P^1(X))} \tau_X^1(P)[Q] \star Q(\tau_P^1(X))$$

Démonstration. Par induction sur P :

$\boldsymbol{P = \{ft_1 \ldots t_k := t_0; s\}}$

Dans ce cas d'une part $\tau^1_X(P) = \{s\}$ et $\tau^1_P(X) = X \oplus (f, \overline{t_1}^X, \ldots \overline{t_k}^X, \overline{t_0}^X)$

Et d'autre part $P[Q] = \{ft_1 \ldots t_k := t_0; s_Q\}\{s\}[Q]$, d'où :

$$\begin{aligned} P[Q] \star X &\succ \{s_Q\}\{s\}[Q] \star X \oplus (f, \overline{t_1}^X, \ldots \overline{t_k}^X, \overline{t_0}^X) \\ &= Q\ \tau^1_X(P)[Q] \star \tau^1_P(X) \\ &\succ_{time(Q, \tau^1_P(X))} \tau^1_X(P)[Q] \star Q(\tau^1_P(X)) \\ &\quad \text{(par commutation de contexte)} \end{aligned}$$

$\boldsymbol{P = \{\texttt{if}\ F\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}}$

Dans ce cas d'une part :

$\tau^1_X(P) = \{s_i; s\}$, où $i = 1$ si $\overline{F}^X = true$ et $i = 2$ sinon, et $\tau^1_P(X) = X$

Et d'autre part :

$P[Q] = \{\texttt{if}\ F\ \texttt{then}\ \{s_Q\}\{s_1\}[Q]\ \texttt{else}\ \{s_Q\}\{s_2\}[Q];\}\{s\}[Q]$, d'où :

$$\begin{aligned} P[Q] \star X &\succ \{s_Q\}\{s_i\}[Q]\{s\}[Q] \star X \\ &= Q\ \tau^1_X(P)[Q] \star \tau^1_P(X) \\ &\quad \text{(d'après la remarque p.98)} \\ &\succ_{time(Q, \tau^1_P(X))} \tau^1_X(P)[Q] \star Q(\tau^1_P(X)) \\ &\quad \text{(par commutation de contexte)} \end{aligned}$$

$\boldsymbol{P = \{\texttt{while}\ F\ \{s_1\}; s\}}$

Alors deux cas :

F est vraie dans X

Dans ce cas, d'une part $\tau^1_X(P) = \{s_1; \texttt{while}\ F\ \{s_1\}; s\}$ et $\tau^1_P(X) = X$

Et d'autre part $P[Q] = \{\texttt{while}\ F\ \{s_Q\}\{s_1\}[Q]; s_Q\}\{s\}[Q]$, d'où :

$$\begin{aligned} P[Q] \star X &\succ \{s_Q\}\{s_1\}[Q]\{\texttt{while}\ F\ \{s_Q\}\{s_1\}[Q]; s_Q\}\{s\}[Q] \star X \\ &= Q\ \tau^1_X(P)[Q] \star \tau^1_P(X) \\ &\quad \text{(d'après la remarque p.98)} \\ &\succ_{time(Q, \tau^1_P(X))} \tau^1_X(P)[Q] \star Q(\tau^1_P(X)) \\ &\quad \text{(par commutation de contexte)} \end{aligned}$$

F est fausse dans X

Dans ce cas, d'une part $\tau^1_X(P) = \{s\}$ et $\tau^1_P(X) = X$

Et d'autre part $P[Q] = \{\texttt{while}\ F\ \{s_Q\}\{s_1\}[Q]; s_Q\}\{s\}[Q]$, d'où :

$$\begin{aligned} P[Q] \star X &\succ \{s_Q\}\{s\}[Q] \star X \\ &= Q\ \tau^1_X(P)[Q] \star \tau^1_P(X) \\ &\quad \text{(d'après la remarque p.98)} \\ &\succ_{time(Q, \tau^1_P(X))} \tau^1_X(P)[Q] \star Q(\tau^1_P(X)) \\ &\quad \text{(par commutation de contexte)} \end{aligned}$$

$\boldsymbol{P = \{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s\}}$

Alors deux cas :

$i < n$ et F est fausse dans X

Dans ce cas, d'une part :

$\tau^1_X(P) = \{s_1; \texttt{loop}\ n\ \texttt{except}\ F\ \{s_1\}; s\}$ et $\tau^1_P(X) = X \oplus (i, \overline{i+1}^X)$

Et d'autre part :

$P[Q] = \{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_Q\}\{s_1\}[Q]; s_Q\}\{s\}[Q]$, d'où :

$$\begin{aligned} P[Q] \star X \succ\ & \{s_Q\}\{s_1\}[Q] \\ & \{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_Q\}\{s_1\}[Q]; s_Q\} \\ & \{s\}[Q] \star X \oplus (i, \overline{i+1}^X) \\ =\ & Q\ \tau_X^1(P)[Q] \star \tau_P^1(X) \\ & \quad \text{(d'après la remarque p.98)} \\ \succ_{time(Q,\tau_P^1(X))}\ & \tau_X^1(P)[Q] \star Q(\tau_P^1(X)) \\ & \quad \text{(par commutation de contexte)} \end{aligned}$$

$i = n$ ou F est vraie dans X

Dans ce cas, d'une part $\tau_X^1(P) = \{s\}$ et $\tau_P^1(X) = X \oplus (i, \overline{0})$

Et d'autre part :

$P[Q] = \{\texttt{loop}\ n\ \texttt{except}\ F\ \{s_Q\}\{s_1\}[Q]; s_Q\}\{s\}[Q]$, d'où ;

$$\begin{aligned} P[Q] \star X \succ\ & \{s_Q\}\{s\}[Q] \star X \oplus (i, \overline{0}) \\ =\ & Q\ \tau_X^1(P)[Q] \star \tau_P^1(X) \\ & \quad \text{(d'après la remarque p.98)} \\ \succ_{time(Q,\tau_P^1(X))}\ & \tau_X^1(P)[Q] \star Q(\tau_P^1(X)) \\ & \quad \text{(par commutation de contexte)} \end{aligned}$$

□

Annexe B

Compléments

Sommaire

B.1 Les structures de données 139
Typage des états 139
Représentation d'un élément 142
Bestiaire des types usuels 143
B.2 Une simulation plus équitable 147
n-uplet de mises à jour 147
Conditionnelles gratuites 149
Boucles gratuites 151
Accélération finale 153
Le modèle « augmenté » 155

B.1 Les structures de données

Dans [GV10] Serge Grigorieff et Pierre Valarcher ont montré l'intérêt du typage dans la présentation des ASMs pour caractériser les modèles de calcul. Ici notre but est simplement de développer la formalisation de la logique du premier ordre faite à la section 1.1 afin d'inclure plus précisément la notion de typage. Nous le faisons pour deux raisons :

1. Comme nous travaillons sur les classes en temps p.33 et les classes en espace p.74 nous avons besoin d'une notion de taille d'éléments. Dans cette section nous définissons la taille d'un élément comme la taille de sa représentation (formée de constructeurs), et vérifions que cette définition a du sens pour les types usuels.
2. Bien que nous n'utilisions réellement que les booléens et les entiers unaires, nous avons tenu à esquisser un bestiaire des types usuels p.143 pour montrer à la fois les possibilités de formalisation offertes par la formalisation de Gurevich, mais aussi pour montrer qu'il est possible de se restreindre à des types usuels sans impacter notre théorème p.103.

Typage des états

À la section 1.3 p.29 nous avons séparé le langage de l'algorithme A entre :

1. $Dyn(A)$: Les symboles dynamiques.
2. $Init(A)$: Les symboles initiaux.

3. $Cons(A)$: Les constructeurs.
4. $Oper(A)$: Les opérations.

Dans cette section nous avons utilisé l'ensemble $Cons(A) \sqcup Oper(A)$ mais nous n'avons pas encore différencié $Cons(A)$ et $Oper(A)$. Comme annoncé, les constructeurs serviront à construire les types, contrairement aux opérations (ou oracles) du langage.

Plus précisément, un type sera une structure dont le langage est composé uniquement de constructeurs. Une base de données sera donc une réunion de types, et un état sera donc une base de donnée enrichie (voir p.54) par les symboles (notamment l'égalité) de $Dyn(A) \sqcup Init(A) \sqcup Oper(A)$:

Définition B.1.1. (Réunion de structures)
Soit $T_1, \ldots, T_k$ ($k \geq 2$) des structures de langage et d'univers disjoints[1] tels que T_1 soit le type $Undef$ p.143 et que l'un de ces types soit $\mathbb{B}$ p.143.
La structure de données $D =_{def} T_1 \sqcup \cdots \sqcup T_k$ est la structure définie par :

1. $\mathcal{U}(D) =_{def} \mathcal{U}(T_1) \sqcup \cdots \sqcup \mathcal{U}(T_k)$
2. $\mathcal{L}(D) =_{def} \mathcal{L}(T_1) \sqcup \cdots \sqcup \mathcal{L}(T_k)$
3. si $f \in F_\alpha(\mathcal{L}(T_i))$ alors $\overline{f}^D$ est définie par :
   - $\overline{f}^D(a_1, \ldots, a_\alpha) =_{def} \overline{f}^{T_i}(a_1, \ldots, a_\alpha)$ si tous les a_i sont dans $\mathcal{U}(T_i)$
   - $\overline{f}^D(a_1, \ldots, a_\alpha) =_{def} \overline{undef}^{T_1}$ sinon.

Remarque. Il est possible par des copies isomorphes des univers et des annotations du langage d'utiliser plusieurs occurrences du même type.

Lemme B.1.2. *(Isomorphisme par réunion)*
Si $D_1 = X_1 \sqcup \cdots \sqcup X_m$ est isomorphe à une structure D_2
Alors $D_2 = Y_1 \sqcup \cdots \sqcup Y_m$ telle que chaque X_i est isomorphe à Y_i.

Démonstration. Soit ζ l'isomorphisme entre D_1 et D_2.
Notons $\mathcal{L}$ le langage de D_1 et $\mathcal{L}_i$ le langage de X_i.
Par isomorphisme D_1 et D_2 ont le même langage $\mathcal{L} = \mathcal{L}_1 \sqcup \cdots \sqcup \mathcal{L}_k$.
Comme la réunion est disjointe chaque symbole f appartient à un seul $\mathcal{L}_i$.
Pour chaque X_i, $\mathcal{U}(X_i) \subseteq \mathcal{U}(D_1)$, donc $\zeta(\mathcal{U}(X_i))$ est bien défini.
Notons Y_i la structure :

1. de langage $\mathcal{L}_i$
2. d'univers $\zeta(\mathcal{U}(X_i))$
3. si $f \in F_\alpha(\mathcal{L}_i)$ alors :
   $\overline{f}^{Y_i}(\zeta(a_1), \ldots, \zeta(a_\alpha)) =_{def} \zeta(\overline{f}^{D_1}(a_1, \ldots, a_\alpha))$

ζ est surjective sur $\mathcal{U}(Y_i)$.
De plus, $\overline{f}^D(a_1, \ldots, a_\alpha) = \overline{f}^{X_i}(a_1, \ldots, a_\alpha)$ si tous les a_i sont dans $\mathcal{U}(X_i)$.
Donc ζ est un isomorphisme entre X_i et Y_i.
Comme $\mathcal{U}(D_1) = \mathcal{U}(X_1) \sqcup \cdots \sqcup \mathcal{U}(X_k)$ et que ζ est bijective nous avons :
$\mathcal{U}(D_2) = \mathcal{U}(Y_1) \sqcup \cdots \sqcup \mathcal{U}(Y_k)$
Enfin, par isomorphisme :
$\overline{f}^{D_1}(a_1, \ldots, a_\alpha) = \overline{undef}^{X_1} \Leftrightarrow \overline{f}^{D_2}(\zeta(a_1), \ldots, \zeta(a_\alpha)) = \overline{undef}^{Y_1}$
Donc $D_2 = Y_1 \sqcup \cdots \sqcup Y_m$. □

1. ↑ Nous pouvons toujours nous ramener au cas où les ensembles de base sont disjoints en prenant des copies isomorphes, et rendre les langages disjoints en les annotant.

Définition B.1.3. (Typage des états)
Nous dirons que $X \in S(A)$ est typé, et que ses types sont $T_1, \ldots, T_k$, si $X|_{Cons(A)} = T_1 \sqcup \cdots \sqcup T_k$.

Comme annoncé p.21, nous montrons que cette notion de typage est bien compatible avec la stabilité par isomorphisme :

Lemme B.1.4. *(Isomorphisme par typage)*
Si un état X est typé et est isomorphe à Y
Alors Y est un état avec les mêmes types, à isomorphisme près.

Démonstration. D'après le second postulat p.24, comme X est un état et que X et Y sont isomorphes alors Y est aussi un état.
Par isomorphisme, X et Y ont le même langage :
$\mathcal{L}(A) = Dyn(A) \sqcup Init(A) \sqcup Oper(A) \sqcup Cons(A)$
D'après le lemme 2.3.3 p.54, comme X et Y sont isomorphes alors $X|_{Cons(A)}$ et $Y|_{Cons(A)}$ sont isomorphes.
Or par hypothèse X a pour types $T_1, \ldots, T_k$.
Donc $X|_{Cons(A)} = T_1 \sqcup \cdots \sqcup T_k$.
Donc d'après le lemme B.1.2 : $Y|_{Cons(A)} = T'_1 \sqcup \cdots \sqcup T'_k$, où chaque T'_i est isomorphe à T_i. □

En fait, comme les interprétations de $Oper(A) \sqcup Cons(A)$ sont uniformes (voir p.29), il suffit de supposer qu'un état est typé pour que tous le soient :

Lemme B.1.5. *(Uniformité des types)*
Si un état est typé alors tous les états sont typés.
De plus, les types sont identiques d'un état à l'autre, à isomorphisme près.

Démonstration. Les interprétations de $Oper(A) \sqcup Cons(A)$ sont uniformes, c'est-à-dire que pour tout $X, Y \in S(A)$:
$X|_{Oper(A) \sqcup Cons(A)}$ est isomorphe à $Y|_{Oper(A) \sqcup Cons(A)}$.
$X|_{Cons(A)}$ et $Y|_{Cons(A)}$ sont des restrictions de ces deux structures.
Donc d'après le lemme 2.3.3 p.54 $X|_{Cons(A)}$ et $Y|_{Cons(A)}$ sont isomorphes.
Or par hypothèse il existe un état X ayant pour types $T_1, \ldots, T_k$.
Donc : $X|_{Cons(A)} = T_1 \sqcup \cdots \sqcup T_k$
D'où d'après le lemme B.1.2 : $Y|_{Cons(A)} = T'_1 \sqcup \cdots \sqcup T'_k$, où chaque T'_i est isomorphe à T_i. □

Ainsi nous pouvons parler des types de l'algorithme, à isomorphisme près.

Remarque. Quand nous enrichissons une structure de données par des symboles d'opérations il n'est pas nécessaire de respecter strictement le typage.
Par exemple nous pouvons ajouter les opérations arithmétiques $+$ et $\times$ de façon à ce qu'elles puissent prendre des entiers unaires et donner des entiers unaires, mais aussi prendre des entiers binaires et donner des entiers binaires.
Cela peut être vu comme notre version du **polymorphisme**.

Toutefois, dans une présentation comme celle de [GV10] il faut faire très attention avec les fonctions polymorphes ou permettant de traduire un type dans un autre.

Représentation d'un élément

Définition B.1.6. (Représentation unique)

Nous dirons qu'un état $X \in S(A)$ est **représentable** si pour tout élément $a \neq \overline{undef}^X$ il existe un unique terme t_a formé uniquement de symboles de $Cons(A)$ tel que $\overline{t_a}^X = a$.

t_a sera appelé la **représentation** de a.

Nous montrerons dans la sous-section suivante qu'en particulier les états typés avec les types usuels sont bien représentables.

Comme annoncé, nous utilisons la notion de représentation d'un élément pour déterminer « naturellement » la taille d'un élément :

Définition B.1.7. (Taille d'un élément)

La taille $|t|$ d'un terme $t \in Term(\mathcal{L})$ est le nombre de symboles le formant :

1. si $t \in F_0(\mathcal{L})$ alors $|t| =_{def} 1$
2. si $t = ft_1 \dots t_\alpha$ où $f \in F_\alpha(\mathcal{L})$ avec $\alpha > 0$ et $t_1, \dots, t_\alpha \in Term(\mathcal{L})$ alors $|t| =_{def} 1 + \sum_{1 \leq i \leq \alpha} |t_i|$

La taille $|a|$ d'un élément $a \in \mathcal{U}(X)$ est la taille de sa représentation t_a.

La taille de $\overline{undef}^X$ est posée à 1 par homogénéité.

Remarque. Si f est un constructeur alors : $|\overline{f}^X(\overline{t_1}^X, \dots, \overline{t_\alpha}^X)| = 1 + \sum_{i=1}^{\alpha} |\overline{t_i}^X|$

En effet pour tout i : $\overline{t_i}^X \in \mathcal{U}(X)$, donc il a une représentation $t_{\overline{t_i}^X}$.

Comme f est un constructeur, $ft_{\overline{t_1}^X} \dots t_{\overline{t_\alpha}^X}$ est la représentation de $\overline{f}^X(\overline{t_1}^X, \dots, \overline{t_\alpha}^X)$.

Or par définition $|ft_{\overline{t_1}^X} \dots t_{\overline{t_\alpha}^X}| = 1 + \sum_{1 \leq i \leq \alpha} |t_{\overline{t_i}^X}|$

Et comme $|t_{\overline{t_i}^X}| = |\overline{t_i}^X|$, nous avons l'égalité recherchée.

Lemme B.1.8. *(Conservation de la taille par isomorphisme)*

Si X est représentable et que X et Y sont isomorphes par ζ
alors Y est représentable également, et pour tout $a \in \mathcal{U}(X)$: $t_{\zeta(a)} = t_a$.
En particulier, pour tout $a \in \mathcal{U}(X)$: $|\zeta(a)| = |a|$.

Démonstration. Par isomorphisme X et Y ont le même langage donc les mêmes termes.

Comme X est représentable, pour tout $a \in \mathcal{U}(X)$ il existe une unique représentation t_a telle que $\overline{t_a}^X = a$.

Or $\zeta(\overline{t_a}^X) = \overline{t_a}^Y$, donc $\overline{t_a}^Y = \zeta(a)$.

Supposons qu'il existe un terme $t \neq t_a$ formé uniquement de constructeurs tel que $\overline{t}^Y = \zeta(a)$.

Dans ce cas $a = \zeta^{-1}(\zeta(a)) = \zeta^{-1}(\overline{t}^Y) = \overline{t}^X$, ce qui contredit l'unicité de t_a pour X.

Donc t_a est bien la représentation de $\zeta(a)$ dans Y, et $|\zeta(a)| = |t_a| = |a|$.

De plus $\overline{undef}^Y = \zeta(\overline{undef}^X)$ d'où $|\zeta(\overline{undef}^X)| = 1 = |\overline{undef}^X|$ □

Corollaire B.1.9. *Deux exécutions isomorphes ont la même classe en temps.*

Démonstration. Soient deux états X et Y isomorphes.

La taille d'un état est définie p.32 par :

$$|X| =_{def} (|f|_X)_{f \in Dyn(A) \sqcup Init(A)}$$

$$|f|_X =_{def} sup_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$$

Donc comme des éléments isomorphes ont la même taille nous avons :

$$|X| = |Y|$$

Le temps d'exécution est donc :

$$\begin{aligned} c_A(|X|) &= sup_{|X|=|Z|} time(A, Z) \\ &= sup_{|Y|=|Z|} time(A, Z) \\ &= c_A(|Y|) \end{aligned}$$

□

Mais ce résultat n'est guère intéressant puisque les deux premiers postulats permettent de prouver directement le lemme 1.2.2 p.25.

Bestiaire des types usuels

Les booléens et les entiers unaires ont déjà été esquissés à la section 1.1.

Nous montrons la preuve suivante tout simplement d'après l'exemple, en développant comment représenter les types usuels dans le cadre de notre formalisme :

Proposition B.1.10. *(Représentabilité des types usuels)*
Les types suivants sont représentables :

1. *Les booléens.*
2. *Les entiers dans une base $b \in \mathbb{N}^\star$ quelconque.*
3. *Les listes (pile ou file) sur un type donné.*
4. *Les tableaux (éventuellement multidimensionnels, associatifs ou dynamiques).*
5. *Les arbres et les graphes.*

Type. $(Undef)$
Nous avons supposé à la définition B.1.1 p.140 que *undef* (voir les fonctions partielles p.16) formait à lui seul un type. Ce n'est pas forcément une présentation intuitive, mais le considérer comme un type est uniquement un moyen de rendre la présentation (univers et langages disjoints) plus homogène.

Type. (Les booléens $\mathbb{B}$)
L'univers $\mathcal{U}(\mathbb{B})$ contient deux éléments distincts.
Les constructeurs sont $\mathcal{L}(\mathbb{B}) = \{true, false\}$, interprétés par $\overline{true}^{\mathbb{B}} \neq \overline{false}^{\mathbb{B}}$.
De plus, nous considérons que nous enrichissons toujours le langage avec les opérations $\{\neg, \wedge\}$ définies p.17, de la même façon que nous considérons toujours que nos langages sont égalitaires (voir p.18).

Type. (Les entiers unaires $\mathbb{N}_1$)
L'univers $\mathcal{U}(\mathbb{N}_1)$ est une copie de $\mathbb{N}$
Les constructeurs sont $\mathcal{L}(\mathbb{N}_1) = \{\emptyset, S\}$, interprétés par :

1. $\overline{\emptyset}^{\mathbb{N}_1} = 0$
2. $\overline{S}^{\mathbb{N}_1} : x \mapsto x + 1$

En particulier, pour les entiers unaires les termes formés uniquement de constructeurs sont de la forme $S^n\emptyset$, et ils sont interprétés par $\overline{S^n\emptyset}^{\mathbb{N}_1} = n$.
Ainsi, comme annoncé p.19, la taille (voir la définition B.1.7 p.142) d'un entier unaire n est alors de $n + 1$.

Type. ($\mathbb{N}_b$, les entiers dans une base entière $b \geq 2$)

L'univers $\mathcal{U}(\mathbb{N}_b)$ est une copie de $\mathbb{N}$.

Les constructeurs sont $\mathcal{L}(\mathbb{N}_b) = \{c_0, \ldots, c_{b-1}, f_0, \ldots, f_{b-1}\}$, interprétés par :

1. $\overline{c_i}^{\mathbb{N}_b} =_{def} i$
2. $\overline{f_0}^{\mathbb{N}_b}(\overline{c_i}^{\mathbb{N}_b}) =_{def} \overline{undef}^X$
   $\overline{f_i}^{\mathbb{N}_b} : a \mapsto a \times b + i$ sinon

Les termes formés uniquement de constructeurs sont de la forme $f_{i_0} \ldots f_{i_{n-1}} c_n$, que nous noterons $\underline{i_n \ldots i_0}_b$.

Il s'agit de la représentation en base b de $\overline{\underline{i_n \ldots i_0}_b}^{\mathbb{N}_b}$.

Par exemple, en base décimale 314 s'écrit $(3 \times 10 + 1) \times 10 + 4$ donc est représentable par $f_4 f_1 c_3 = \underline{314}_{10}$.

La taille de $a \in \mathcal{U}(\mathbb{N}_b)$ est le nombre de chiffres de sa représentation. Par exemple $|314| = |\overline{\underline{314}_{10}}^{\mathbb{N}_{10}}| = |\overline{f_4 f_1 c_3}^{\mathbb{N}_{10}}| = 3$.

Remarque. Sur les entiers :

1. Nous aurions pu relâcher la contrainte $\overline{f_0}^{\mathbb{N}_b}(\overline{c_i}^{\mathbb{N}_b}) = \overline{undef}^X$ en $\overline{f_0}^{\mathbb{N}_b}(\overline{c_0}^{\mathbb{N}_b})$ $= \overline{undef}^X$ de façon à générer les chiffres $1, \ldots, d-1$ avec $f_i c_0$ au lieu de devoir utiliser les symboles de constantes $c_1, \ldots, c_{b-1}$. Le problème est qu'alors $1, \ldots, d-1$ seraient considérés comme les nombres à deux chiffres et non à un chiffre comme attendu.
2. La contrainte $\overline{f_0}^{\mathbb{N}_b}(\overline{c_0}^{\mathbb{N}_b}) = \overline{undef}^X$ en revanche est nécessaire pour empêcher d'avoir une infinité de représentants $\underline{0}_b$, $\underline{00}_b$, $\underline{000}_b$, etc. pour 0.
3. Ces deux problèmes expliquent qu'à part pour les nombres à un chiffre, il n'y a pas de nombres commençant par 0. Il est toutefois possible de se passer du 0 avec un système de numération bijectif utilisant le langage $\{\emptyset, f_1, \ldots, f_b\}$, bien que ce ne soit pas une présentation usuelle.

   En particulier, les entiers unaires correspondent à ce système pour $b = 1$, alors qu'il n'est pas possible d'avoir une base 1 dans la présentation usuelle.

Type. (Les listes de type T)

Nous supposerons que T est un type déjà défini, et nous montrons comment définir les piles et les files sur T.

L'univers est une copie des suites finies d'éléments de l'univers de T.

Les constructeurs sont $\{\epsilon, prepend\}$ pour les piles et $\{\epsilon, append\}$ pour les files, interprétés par :

1. $\overline{\epsilon}^X =_{def} ()$
2. $\overline{prepend}^X(a_0, (a_1, \ldots, a_n)) =_{def} (a_0, a_1, \ldots, a_n)$
3. $\overline{append}^X((a_1, \ldots, a_n), a_{n+1}) =_{def} (a_1, \ldots, a_n, a_{n+1})$

Les deux types sont naturellement enrichis par des opérations *head* et *tail* interprétées uniformément par :

1. $\overline{head}^X((a_0, a_1, \ldots, a_n)) =_{def} a_0$
2. $\overline{tail}^X((a_0, a_1, \ldots, a_n)) =_{def} (a_1, \ldots, a_n)$

Une pile (a, b, c) sera représentée par :

$$prepend(t_a, prepend(t_b, prepend(t_c, \epsilon)))$$

alors qu'une file (a, b, c) sera représentée par :

$$append(append(append(\epsilon, t_a), t_b), t_c)$$

Dans les deux cas, la taille de (a, b, c) sera $|t_a| + |t_b| + |t_c| + 4$.
Plus généralement, la taille d'une liste $\vec{a}$ est :

$$|(\vec{a})| = 1 + length(\vec{a}) + \sum_{a \in \{\vec{a}\}} |a|$$

Remarque. Sur les listes :

1. Généralement une relation $IsEmpty$ est demandée dans ce genre de structure. Nous l'avons immédiatement en utilisant ϵ et l'égalité.
2. Pour avoir une représentation unique, *prepend* et *append* ne peuvent être utilisées en même temps comme constructeur. En revanche, si l'une est choisie comme constructeur, l'autre peut sans problème être utilisée comme opération. Ce choix n'a en fait pas d'importance, puisque les deux conduisent à la même taille pour les éléments.
3. S'il existe une borne K à la taille des éléments d'une liste, alors :

$$\begin{aligned} |(\vec{a})| &= 1 + length(\vec{a}) + \textstyle\sum_{a \in \{\vec{a}\}} |a| \\ &\leq 1 + length(\vec{a}) + \textstyle\sum_{a \in \{\vec{a}\}} K \\ &= 1 + length(\vec{a}) \times (1 + K) \\ &= O(length(\vec{a})) \end{aligned}$$

   Ce qui correspond à l'intuition habituelle que la taille d'une liste est de l'ordre de sa longueur.

Type. (Les tableaux)

Un tableau est implémenté dans notre présentation par une fonction dynamique *tab*. Notamment, il est multi-dimensionnel si son arité vérifie $\alpha > 1$.

$Dom(tab, X)$ peut être une partie de $\mathbb{N}_1^\alpha$ dans le cas des tableaux indicés, sinon il s'agit plus généralement de tableaux associatifs (prenons par exemple l'annuaire, qui associe un numéro de téléphone à un nom).

Nous avons défini p.32 la taille d'un tableau comme étant :

$$|f|_X =_{def} \sup_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$$

Ainsi, la taille d'un tableau est en fait la taille maximale d'une case, ce qui peut surprendre puisque l'usage est plutôt de donner la longueur pour la taille.

Le problème est que, dans notre présentation, la longueur d'un tableau n'est pas forcément finie, ce qui correspond à la notion de tableau dynamique (parfois appelé vecteur).

Toutefois il ne faut pas craindre pour autant que le tableau puisse contenir une infinité de valeurs. En effet, comme le langage est fini il n'y a qu'un nombre fini de représentations d'une taille donnée. Donc si la taille d'une case est bornée alors le tableau ne contient qu'un nombre fini de représentations. Ainsi, si les états sont représentables alors il y a un nombre fini de valeurs potentielles.

Nous avons donc naturellement que la quantité d'information portée par le tableau est finie, et dépend bien de sa taille. Dit autrement, il n'y a pas de limite a priori au nombre d'adresses virtuelles mais le nombre d'adresses physiques est borné, le tableau pouvant alors être vu comme une structure complexe de pointeurs sur une mémoire physique donnée.

Si nous souhaitons un tableau de longueur finie $\ell_{tab} \in Init(A)$, il est possible d'imposer que $tab(i) = undef$ partout dans l'état initial, sauf où $0 \leq i < \ell_{tab}$. Pour maintenir la longueur du tableau durant l'exécution, il est possible de remplacer les mises à jour $tab(i) := t$ par `if` $0 \leq i < \ell_{tab}$ `then` $tab(i) := t$ ou directement interdire ce genre de mises à jour dans la sémantique opérationnelle.

Dans ce cas, le fait que $\ell_{tab} \in Init(A)$ permet de parcourir les cases du tableau dans le langage $\texttt{LoopC}_{\texttt{stat}}$ défini p.81.

Remarque. Si la taille d'un tableau devait être :

$$|f|_X =_{def} \sum_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$$

Dans ce cas il pourrait être légitime de poser que $|\overline{undef}^X| = 0$ afin qu'un tableau ayant un domaine fini ait une taille finie.

Ces deux mesures tiennent compte des valeurs du tableau, conformément à notre remarque p.31 sur les valeurs « explosives ». Toutefois, dans un tri habituel comme le tri rapide seule la longueur du tableau compte et non ses valeurs. Ainsi, en un sens, notre mesure serait peut-être plus adaptée à un tri comptage.

Toutefois, dans une implémentation du tri rapide la longueur du tableau, ou plutôt les indices de début et de fin du tableau, sont passés en arguments donc font partie de la taille de l'espace au sens où nous l'avons défini. Les valeurs du tableau sont ignorées car seules des comparaisons et des échanges de valeur sont effectuées, et non des calcul utilisant ces valeurs. Ainsi, c'est davantage une spécificité de l'algorithme qu'un critère général qui détermine la mesure de complexité dans ce cas.

Type. (Les graphes)

Un graphe est la donnée de deux ensembles (V, E) où V est l'ensemble des sommets et $E \subseteq V^2$ est l'ensemble des arêtes. De plus, si le graphe n'est pas simple, il faut ajouter une fonction d'incidence γ.

Conformément à notre remarque p.18, nous prendrons des fonctions indicatrices pour V et E, ce qui revient à disposer de la matrice d'adjacence du graphe. Cette implémentation convient davantage aux graphes denses, mais pour les graphes creux généralement un tableau est utilisé pour relier l'étiquette d'un sommet à la liste des voisins.

Un arbre (graphe orienté connexe et acyclique) en particulier est un graphe, donc peut être représenté comme énoncé ci-dessus. Sinon il existe également des représentations propres, comme :

1. Des listes de listes, par exemple (L, R) où L est le sous-arbre gauche et R le sous-arbre droit, voire en étendant les types $(a, (L, R))$ où a est l'étiquette du nœud en cours.

2. Un tableau avec des contraintes d'indexage, par exemple l'indice i en binaire correspondant au cheminement dans un arbre binaire, ou alors les arbres binaires de recherche.

B.2 Une simulation plus équitable

Notre théorème p.103 montre l'équivalence des ASMs en certaines classes en temps avec des fragments particuliers des programmes impératifs. Rappelons que cette notion d'équivalence est définie p.56 et se base sur l'équivalence entre exécutions :

1. à variables temporaires près
2. à dilatation temporelle près

D'ailleurs cette équivalence peut être critiquée car les ASMs arrivent à simuler les programmes impératifs strictement pas à pas et sans temps d'arrêt, contrairement aux programmes impératifs qui ont besoin d'une dilatation temporelle et d'un temps d'arrêt importants et dépendant fortement du programme à simuler.

Ce résultat a fortement influencé notre définition de la simulation, car nous avions besoin des ASMs comme modèle pour les algorithmes séquentiels, et nous voulions un modèle simple des programmes impératifs, afin de pouvoir adapter notre résultat au plus de langages de programmation possibles.

Toutefois il est possible de prendre le point de vue inverse, à savoir chercher comment modifier notre modèle des langages impératifs pour qu'il corresponde le plus possible aux ASMs, cela en étant de plus en plus exigeant dans notre définition de la simulation.

Pour reprendre notre remarque p.104 peut-être que les ASMs sont « plus équivalentes » que nos programmes impératifs actuels, mais que rajouter à notre modèle des programmes impératifs pour que la simulation soit plus équitable ?

Rappel. Pour simuler une ASM Π nous avons utilisé le programme suivant :

$$P_\Pi \; P_{c_\Pi}^{F_\Pi}[\texttt{if } \neg F_\Pi \texttt{ then } P_\Pi \texttt{ else } \{i_{end} := i_{end} + 3;\}] \texttt{ skip } i_{end} \to max_{end}$$

1. Les variables temporaires sont $Var(P_{c_\Pi}) \sqcup \{v_t \mid t \in T(\Pi)\} \sqcup \{i_{end}\}$
2. La dilatation temporelle est $d = t_\Pi + 2$
3. Le temps d'arrêt est $e = t_\Pi + 6 \times (nest(P_{c_\Pi}) + 1) + 1$

$t_\Pi = r + c + m$ est obtenu à la proposition p.89 tel que :
- r est le nombre de termes lus par Π (voir p.37)
- c est le nombre d'états reconnus par Π (voir p.40)
- m est le degré de parallélisme de Π (voir p.89)

> Attention : Dans ce chapitre nous ne faisons que donner des idées pour rendre notre simulation plus « équitable ». Certaines de ces idées feront probablement l'objet de travaux futurs, mais nous nous contentons ici de les esquisser dans un but illustratif, et non de les prouver.

n-uplet de mises à jour

Ce qui a posé problème durant la traduction est que les ASMs ont un comportement parallèle [1] contrairement à nos programmes impératifs qui étaient strictement séquentiels. Toutefois certains langages possèdent un peu de parallélisme, par exemple des commandes de la forme $(x, y) := (y, x)$ qui correspondent exactement à notre exemple p.86.

En suivant cette idée nous pouvons autoriser des mises à jour de la forme :

$$(f_1\vec{t}_1, \ldots, f_m\vec{t}_m) := (t_1, \ldots, t_m)$$

1. ↑ Mais borné, d'après le troisième postulat p.27.

et dont la sémantique opérationnelle serait :

$$\{(f_1\vec{t_1}, \ldots, f_m\vec{t_m}) := (t_1, \ldots, t_m); s\} \star X$$
$$\succ \{s\} \star X \oplus \{(f_1, \vec{t_1}^X, \overline{t_1}^X), \ldots, (f_m, \vec{t_m}^X, \overline{t_m}^X)\}$$

Ainsi, les $\{v_t \mid t \in T(\Pi)\}$ ne sont plus nécessaire pour assurer la parallélisation des mises à jour, celle-ci est désormais disponible « nativement » dans le modèle. Toutefois, elles restent utilisées dans la μ-formule de Π :

Rappel.

$$F_\Pi =_{def} \bigwedge_{t \in Read(\Pi)} v_t = t$$

Cette formule est utilisée pour détecter l'arrêt de l'exécution au lemme p.94 :

$$min\{i \in \mathbb{N} \mid \overline{F_\Pi}^{P_\Pi^{i+1}(X)} = true\} = time(\Pi, X|_{\mathcal{L}(\Pi)})$$

Cependant cette formule n'est pas nécessaire.

En effet, si l'ASM est sous forme normale, nous avons que :

$$X \in S(\Pi) \text{ est terminal } \Leftrightarrow \tau_\Pi(X) = X \Leftrightarrow \Delta(\Pi, X) = \varnothing$$

Donc l'ASM est terminale si et seulement s'il existe un bloc de mises à jour `par endpar` dans la forme normale. Ainsi, comme dans [MV09] nous pouvons le traduire explicitement par $b_{end} := true$ et remplacer F_Π par b_{end} comme condition d'arrêt dans tout le programme final.

Ainsi, au lieu des variables temporaires $\{v_t \mid t \in T(\Pi)\}$ nous n'utiliserons qu'une variable b_{end} initialisée à *false*.

Par commodité, si l'ASM admet une exécution terminale, nous pouvons échanger la position des blocs pour supposer que c'est le dernier qui est vide. Si elle n'admet aucune exécution terminale alors le dernier bloc sera traduit normalement à la place.

Nous obtenons donc à la table B.1 p.148 un nouveau programme P_Π pour traduire une étape d'ASM sous forme normale.

Table B.1 – Traduction avec conditionnelles

```
        if F_1
            (f_1^1 t_1^1, ..., f_{m_1}^1 t_{m_1}^1) := (t_1^1, ..., t_{m_1}^1)
            skip c − 1
else if F_2
            (f_1^2 t_1^2, ..., f_{m_2}^2 t_{m_2}^2) := (t_1^2, ..., t_{m_2}^2)
            skip c − 2
        ⋮
else if F_c
            (f_1^c t_1^c, ..., f_{m_c}^c t_{m_c}^c) := (t_1^c, ..., t_{m_c}^c) ou b_end := true
```

Les conditionnelles imbriquées permettent d'utiliser les F_i et non des v_{F_i}. Toutefois, cela vient avec un coût : il ajouter des `skip` pour homogénéiser la dilatation temporelle, et au final nous avons :

$$time(P_\Pi) = c + 1$$

Cette dilatation temporelle est certes plus faible mais dépend toujours de Π et reste potentiellement gigantesque. En effet $c = B(n)$, c'est-à-dire le n-ième nombre de Bell.

Conditionnelles gratuites

Cela peut être critiquable, mais les ASMs peuvent explorer un nombre quelconque arbitrairement grand de conditionnelles. Pour que la simulation soit équitable, nous supposons donc qu'une conditionnelle peut être évaluée en un temps de 0 :

$$\{\texttt{if } F \ \{s_1\} \texttt{ else } \{s_2\}; s_3\} \star X \succ_0 \{s_i; s_3\} \star X$$

où $i = 1$ si F est vraie, et $i = 2$ si F est fausse.

Nous obtenons donc à la table B.2 p.149 un nouveau programme P_Π.

Table B.2 – Traduction sans conditionnelle

$$\begin{array}{l}
\quad\texttt{if } F_1 \\
\qquad (f_1^1\vec{t}_1^1, \ldots, f_{m_1}^1\vec{t}_{m_1}^1) := (t_1^1, \ldots, t_{m_1}^1) \\
\texttt{else if } F_2 \\
\qquad (f_1^2\vec{t}_1^2, \ldots, f_{m_2}^2\vec{t}_{m_2}^2) := (t_1^2, \ldots, t_{m_2}^2) \\
\quad\vdots \\
\texttt{else if } F_c \\
\qquad (f_1^c\vec{t}_1^c, \ldots, f_{m_c}^c\vec{t}_{m_c}^c) := (t_1^c, \ldots, t_{m_c}^c) \text{ ou } b_{end} := \mathit{true}
\end{array}$$

Et nous avons enfin :

$$time(P_\Pi) = 1$$

Les programmes P_{c_Π} construits à la p.92 n'utilisent pas de conditionnelles donc ils vérifient toujours :

$$time(P_{c_\Pi}, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$$

De plus, ils gardent leurs propriétés, notamment bornes statiques et profondeur dans le cas polynomial.

Remarque. Dans le cas où c_Π est quelconque, il est possible qu'aucune exécution ne soit terminale, et donc aucune variable n'est alors nécessaire. Si une exécution est terminale, seule b_{end} est nécessaire.

Il n'y a que dans le cas primitif récursif que de nouvelles variables sont ajoutées, et dans ce cas le nombre de variables temporaires dépendra de la complexité du programme à simuler.

Dans le cas polynomial, b_{end} sera nécessaire et suffisante.

Nous pouvons garder la même traduction 4.2.2 p.94 pour modifier le programme afin d'inclure la condition d'arrêt. Toutefois, comme les conditionnelles sont devenues gratuites, nous aurons alors que :

$$\text{si } F_{end} \text{ reste fausse alors } time(P^{F_{end}}, X) = time(P, X)$$

Mais cela est suffisant, car le programme coquille $P = P_{c_\Pi}^{b_{end}}$ vérifie :

$$\text{si } F_{end} \text{ reste fausse alors } time(P, X) \geq time(\Pi, X|_{\mathcal{L}(\Pi)})$$

Le problème est que le lemme 4.2.4 p.95 reste applicable, c'est-à-dire que si F_{end} est fausse alors :

$$P_k^{F_{end}}\ P_{k-1}^{F_{end}} \ldots\ P_1^{F_{end}} \succ_1 Q_\ell^{F_{end}}\ Q_{\ell-1}^{F_{end}} \ldots\ Q_1^{F_{end}}$$

$$\text{tel que } \max_{1\leq i\leq k}\{nest(P_i)+i\} \geq \max_{1\leq j\leq \ell}\{nest(Q_j)+j\}$$

Ainsi, jusqu'à ce que b_{end} devienne vraie nous avons :

$$P_{c_\Pi}^{F_\Pi} \star X \succ_i Q_\ell^{F_\Pi} \dots\ Q_1^{F_\Pi} \star X'$$

$$\text{tel que } \ell \leq nest(P_{c_\Pi})+1$$

$$\text{Donc } time(\tau_X^i(P), \tau_P^i(X)) \leq nest(P_{c_\Pi})+1$$

Ainsi, le temps d'arrêt dépend encore de l'état initial, donc il faut utiliser comme programme « cœur » :

$$Q = \texttt{if } \neg b_{end} \texttt{ then } P_\Pi \texttt{ else } i_{end} := i_{end}+2$$

Nous avons seulement besoin d'un +2 et plus d'un +3 car la conditionnelle `if` $\neg b_{end}$ est désormais gratuite.

Attention : comme les conditionnelles de $P = P_{c_\Pi}^{b_{end}}$ sont désormais gratuites il faut changer la définition de l'insertion 4.3.1 p.98 pour ne plus considérer les conditionnelles comme des étapes :

$$\begin{aligned}
\{\}[Q] &=_{def} \{\} \\
\{c;s\}[Q] &=_{def} (c)[Q]\ \{s\}[Q]
\end{aligned}$$

$$\begin{aligned}
(ft_1\dots t_k := t_0)[Q] &=_{def} ft_1\dots t_k := t_0\ Q \\
(\texttt{if } F \texttt{ then } P_1 \texttt{ else } P_2)[Q] &=_{def} \texttt{if } F \texttt{ then } P_1[Q] \texttt{ else } P_2[Q] \\
(\texttt{while } F\ P_1)[Q] &=_{def} \texttt{while } F\ (Q\ P_1[Q]) \\
(\texttt{loop } n \texttt{ except } F\ P_1)[Q] &=_{def} \texttt{loop } n \texttt{ except } F\ (Q\ P_1[Q])
\end{aligned}$$

Nous utilisons donc le programme suivant pour la simulation :

$$P_\Pi\ P_{c_\Pi}^{b_{end}}[\texttt{if } \neg b_{end} \texttt{ then } P_\Pi \texttt{ else } i_{end} := i_{end}+2]\ \texttt{skip}\ i_{end} \to 2\times(nest(P_{c_\Pi})+1)$$

Remarque. Nous avons toujours besoin d'une étape supplémentaire de P_Π, consistant tout simplement à rendre b_{end} vraie quand l'ASM, elle, s'est arrêtée.

Ainsi, nous avons réussi à obtenir une dilatation temporelle uniforme et beaucoup plus faible $\boxed{d=2}$ ce qui est presque une simulation strictement étape par étape (il aurait fallu $d=1$, voir p.55).

En revanche le temps d'arrêt lui est de :

1. Une étape pour rendre b_{end} vraie.
2. $i \leq 2\times(nest(P_{c_\Pi})+1)$ étapes pour finir $P = P_{c_\Pi}^{b_{end}}$.
3. Le matelassage coûtant $2\times(nest(P_{c_\Pi})+1)-i+1$ étapes.

Donc $e = 1+i+2\times(nest(P_{c_\Pi})+1)-i+1$. Nous avons donc un temps d'arrêt de $\boxed{e = 2\times(nest(P_{c_\Pi})+2)}$ étapes.

Comme nous avons déjà rendu les conditionnelles gratuites, l'étape suivante consiste à rendre également les boucles gratuites pour corriger ce problème.

Boucles gratuites

C'est effectivement possible au prix de changements dans les programmes P_{c_Π}, en ajoutant une mise à jour triviale $x := x$ [1] dans le corps des boucles vides afin de tout de même les comptabiliser dans le calcul du temps :

c_Π quelconque

$$P_{c_\Pi} = \{\texttt{while}\ \mathit{true}\ \{x := x;\};\} \in \texttt{While}$$

$c_\Pi \in \mathcal{PR}$

$$P_{c_\Pi} = \{r := P_{\texttt{loop}}(n_1, \ldots, n_k); \texttt{loop}\ r\ \{x := x;\};\} \in \texttt{Loop}$$

$c_\Pi \in \mathcal{P}ol$

Il faut prendre la composition des programmes de la forme donnée à la table B.3 p.151.

Table B.3 – Nouveau programme pour la complexité polynomiale

$$\begin{array}{l}
\texttt{loop}\ n_{d_1,\ldots,d_k} \\
\quad \left.\begin{array}{l}
\texttt{loop}\ n_1 \\
\quad \ddots \\
\qquad \texttt{loop}\ n_1
\end{array}\right\} d_1 \text{ fois} \\
\qquad\qquad \ddots \\
\qquad\qquad\quad \left.\begin{array}{l}
\texttt{loop}\ n_k \\
\quad \ddots \\
\qquad \texttt{loop}\ n_k
\end{array}\right\} d_k \text{ fois} \\
\qquad\qquad\qquad\qquad x := x
\end{array}$$

Comme les conditionnelles et les boucles sont gratuites, seules les mises à jour comptent désormais. Nous n'allons donc plus utiliser l'insertion, mais à la place fusionner :

1. une mise à jour $x := t$ de P_{c_Π} (pouvant être une de nos mises à jour triviale $x := x$ ou une mise à jour non triviale dans le cas primitif récursif)
2. avec la mise à jour $(f_1^i \vec{t}_1^i, \ldots, f_{m_i}^i \vec{t}_{m_i}^i) := (t_1^i, \ldots, t_{m_i}^i)$ de P_Π (dans le cas où c'est F_i qui est vraie)
3. pour obtenir la mise à jour $(x, f_1^i \vec{t}_1^i, \ldots, f_{m_i}^i \vec{t}_{m_i}^i) := (t, t_1^i, \ldots, t_{m_i}^i)$ (rappelons que les symboles de P_Π et P_{c_Π} sont disjoints)

Plus formellement, soit $x := t \uplus P_\Pi$ le programme de la table B.4 p.152.

Ainsi, en substituant dans P_{c_Π} toutes les mises à jour $x := t$ par $x := t \uplus P_\Pi$ nous avons une simulation strictement pas à pas de Π, c'est-à-dire que $\boxed{d = 1}$

Il ne reste qu'à arrêter le programme, c'est-à-dire comme tout est gratuit sauf les mises à jour, empêcher de nouvelles mises à jour de se produire si l'ASM s'est arrêtée. Pour cela deux solutions sont possibles :

1. ↑ Seul l'ASM `par endpar` n'a pas besoin de symboles dynamiques, et elle peut être simulée directement par $\{\}$. Si les symboles dynamiques sont tous d'arité > 0 il suffit de prendre $f(\mathit{true}, \ldots, \mathit{true}) := f(\mathit{true}, \ldots, \mathit{true})$ sans se préoccuper du type, ou de prendre des arguments appropriés si les termes doivent être bien typés.

Table B.4 – Fusion de la mise à jour et de la traduction

$$
\begin{array}{l}
\texttt{if } F_1 \\
\quad (x, f_1^1\vec{t}_1^1, \ldots, f_{m_1}^1\vec{t}_{m_1}^1) := (t, t_1^1, \ldots, t_{m_1}^1) \\
\texttt{else if } F_2 \\
\quad (x, f_1^2\vec{t}_1^2, \ldots, f_{m_2}^2\vec{t}_{m_2}^2) := (t, t_1^2, \ldots, t_{m_2}^2) \\
\vdots \\
\texttt{else if } F_c \\
\quad (x, f_1^c\vec{t}_1^c, \ldots, f_{m_c}^c\vec{t}_{m_c}^c) := (t, t_1^c, \ldots, t_{m_c}^c) \text{ ou } (x, b_{end}) := (t, \mathit{true})
\end{array}
$$

1. Faire la substitution dans $P_{c_\Pi}^{b_{end}}$ et non P_{c_Π}.
   Dans ce cas quand b_{end} devient vraie le bloc en cours est effacé à cause de la conditionnelle `if` $\neg b_{end}$ `then` $\{s\}^{b_{end}};\}$ puis les blocs restants commencent tous par des boucles donc sont également effacés gratuitement.
2. Une autre solution consiste plus simplement à substituer en fait à la mise à jour $x := t$ du programme P_{c_Π} le programme $x := t \uplus P_\Pi^{b_{end}}$ obtenu à la table B.5 p.152.

Table B.5 – Fusion avec condition d'arrêt

$$
\begin{array}{l}
\texttt{if } (F_1 \wedge \neg b_{end}) \\
\quad (x, f_1^1\vec{t}_1^1, \ldots, f_{m_1}^1\vec{t}_{m_1}^1) := (t, t_1^1, \ldots, t_{m_1}^1) \\
\texttt{else if } (F_2 \wedge \neg b_{end}) \\
\quad (x, f_1^2\vec{t}_1^2, \ldots, f_{m_2}^2\vec{t}_{m_2}^2) := (t, t_1^2, \ldots, t_{m_2}^2) \\
\vdots \\
\texttt{else if } (F_c \wedge \neg b_{end}) \\
\quad (x, f_1^c\vec{t}_1^c, \ldots, f_{m_c}^c\vec{t}_{m_c}^c) := (t, t_1^c, \ldots, t_{m_c}^c) \text{ ou } (x, b_{end}) := (t, \mathit{true})
\end{array}
$$

La première solution est néanmoins à privilégier, car même si les deux sont théoriquement équivalentes il y a davantage d'opérations sensées être gratuites dans la seconde, ce qui pourrait avoir un impact en pratique.

Dans tous les cas, nous avons bien une simulation strictement étape par étape de l'ASM, et seule est comptée la dernière mise à jour $(x, b_{end}) := (t, \mathit{true})$ pour que le programme impératif puisse s'arrêter. Donc $\boxed{e = 1}$

Remarque. Nous n'avons utilisé des mises à jour triviales $x := x$ que pour avoir une présentation uniforme, mais en réalité dans les cas où c_Π est quelconque ou polynomiale il est possible d'utiliser des mises à jour

$$(f_1^i\vec{t}_1^i, \ldots, f_{m_i}^i\vec{t}_{m_i}^i) := (t_1^i, \ldots, t_{m_i}^i)$$

au lieu des mises à jour :

$$(x, f_1^i\vec{t}_1^i, \ldots, f_{m_i}^i\vec{t}_{m_i}^i) := (x, t_1^i, \ldots, t_{m_i}^i)$$

Ainsi, le degré de parallélisme[1] de la simulation n'est en fait incrémenté que dans le cas primitif récursif.

1. ↑ Nous estimons cette remarque importante car le degré de parallélisme peut conduire à modifier la valeur de la dilatation temporelle, donc si le temps est une ressource alors le degré de parallélisme devrait aussi être considéré comme une ressource. Et d'autant plus qu'elle est limitée d'après le troisième postulat.

En un sens, le cas primitif récursif est donc plus problématique que le cas quelconque ou polynomial, car il nécessite davantage de travail pour établir la fonction de complexité et donc conduit à introduire des variables en plus de b_{end} et à incrémenter le degré de parallélisme.

Accélération finale

Cela peut sembler gourmand mais il existe une technique intéressante quoiqu'un peu complexe permettant de passer à $e = 0$. Bien sûr la question ne se pose que dans le cas où l'ASM est non vide et admet une exécution terminale, donc soit F_c (avec $c > 1$) la conditionnelle contrôlant le bloc vide de Π.

Nous savons (voir p.40) que F_c est de la forme :

$$\bigwedge_{1\leq i,j\leq n} t_i \ \epsilon_{ij}^c \ t_j, \text{ où } \{t_1, \ldots, t_n\} = Sub(T(\Pi)) \text{ et } \epsilon_{ij}^c \in \{=, \neq\}$$

Dans P_Π le bloc de F_c se contente de faire la mise à jour $b_{end} = true$. L'idée est d'enlever le bloc de F_c et de remplacer dans P_Π le bloc de chaque conditionnelle F_k (pour $k \neq c$) par un embranchement donnant sur la mise à jour $(f_1^k \vec{t}_1^k, \ldots, f_{m_k}^k \vec{t}_{m_k}^k) := (t_1^k, \ldots, t_{m_k}^k)$ dans le cas normal, ou la mise à jour $(f_1^k \vec{t}_1^k, \ldots, f_{m_k}^k \vec{t}_{m_k}^k, b_{end}) := (t_1^k, \ldots, t_{m_k}^k, true)$ si nous arrivons à détecter que le prochain état aurait dû vérifier F_c.

Remarque. Donc le coût de la méthode est d'augmenter le nombre de conditionnelles vérifiées par étape (mais elles sont considérées comme gratuites) ainsi que d'augmenter le degré de parallélisme du programme obtenu (mais malgré nos remarques il n'est pas compté dans notre définition de la simulation).

Soit X un état vérifiant F_k. Après le bloc [1] $(f_1 \vec{t}_1, \ldots, f_m \vec{t}_m) := (t_1, \ldots, t_m)$ de F_k l'état suivant est :

$$Y = X \oplus \{(f_1, \vec{t}_1^X, \overline{t_1}^X), \ldots, (f_m, \vec{t}_m^X, \overline{t_m}^X)\}$$

Donc il faudrait déterminer si F_c serait vraie dans Y, c'est-à-dire que :

$$\text{pour tout } 1 \leq i, j \leq n : \overline{t_i}^Y \ \epsilon_{ij}^c \ \overline{t_j}^Y$$

Exemple B.2.1. Supposons que F_c se réduise à $fx = y$.

Si $fx := t$ apparaît dans les mises à jour F_c deviendrait $t = y$.

Mais ce n'est pas aussi simple : si $fz := t$ apparaissait dans les mises à jour tel que $x = z$ dans X, alors F_c deviendrait quand même $t = y$.

Le même argument s'applique également à y.

De façon plus générale, pour tout terme $ft_1 \ldots t_\alpha$ apparaissant dans F_c, si $f\theta_1 \ldots \theta_\alpha := \theta_0$ apparaît dans le bloc B_k de mises à jour de F_k et que pour tout $i : t_i = \theta_i$ dans X alors $ft_1 \ldots t_\alpha$ devrait être remplacé par θ_0.

Or pour tout $i : t_i, \theta_i \in Sub(T(\Pi))$ donc grâce à la formule F_k nous savons si $t_i = \theta_i$ dans X ou non. Nous obtenons donc la formule suivante :

$$F_c^k =_{def} \bigwedge_{1\leq i,j\leq n} (t_i)^k \ \epsilon_{ij}^c \ (t_j)^k$$

$$\text{où } (ft_1 \ldots t_\alpha)^k = \begin{cases} \theta_0 & \text{si } f\theta_1 \ldots \theta_\alpha := \theta_0 \in B_k \\ & \text{et } \forall\ 1 \leq i \leq \alpha\ t_i = \theta_i \in F_k \\ ft_1 \ldots t_\alpha & \text{sinon} \end{cases}$$

1. ↑ Les exposants ont été enlevés pour simplifier la notation.

Remarque. $(f\theta_1 \dots \theta_\alpha)^k$ est bien défini car l'ASM est sous forme normale donc l'ensemble des mises à jour est consistant.

Ainsi, F_c^k est vraie dans X si et seulement si F_c est vraie dans Y, donc en prenant comme P_Π le programme de la table B.6 p.154 nous obtenons encore une simulation de Π, mais cette fois avec $\boxed{e = 0}$

Table B.6 – Accélération finale

```
        if F_1
           if F_c^1
              (f_1^1 t_1^1, ..., f_{m_1}^1 t_{m_1}^1, b_end) := (t_1^1, ..., t_{m_1}^1, true)
           else
              (f_1^1 t_1^1, ..., f_{m_1}^1 t_{m_1}^1) := (t_1^1, ..., t_{m_1}^1)
else if F_2
           if F_c^2
              (f_1^2 t_1^2, ..., f_{m_2}^2 t_{m_2}^2, b_end) := (t_1^2, ..., t_{m_2}^2, true)
           else
              (f_1^2 t_1^2, ..., f_{m_2}^2 t_{m_2}^2) := (t_1^2, ..., t_{m_2}^2)
        ⋮
else if F_{c-1}
           if F_c^{c-1}
              (f_1^{c-1} t_1^{c-1}, ..., f_{m_{c-1}}^{c-1} t_{m_{c-1}}^{c-1}, b_end) := (t_1^{c-1}, ..., t_{m_{c-1}}^{c-1}, true)
           else
              (f_1^{c-1} t_1^{c-1}, ..., f_{m_{c-1}}^{c-1} t_{m_{c-1}}^{c-1}) := (t_1^{c-1}, ..., t_{m_{c-1}}^{c-1})
```

Remarque. Ce procédé fonctionne aussi « simplement» car b_{end} n'apparaît pas dans les blocs de mises à jour en dehors de F_c.

Toutefois, en utilisant la transformation $(t)^k$ définie plus tôt il semble possible de généraliser le procédé pour tous les blocs et pas seulement le dernier, afin de compresser deux étapes en une seule.

Ce procédé nécessite des conditionnelles gratuites et un degré de parallélisme borné mais quelconque. Donc il ne serait pas utilisable par exemple dans `Imp`, mais pourrait l'être dans `ASM`.

Ainsi, je suspecte qu'en le généralisant il soit possible de prouver que pour toute ASM Π il existe une ASM Π^2 de même langage et états telle que :

$$\tau^i_{\Pi^2}(X) = \tau^{2\times i}_{\Pi}(X)$$

Ce serait un argument très fort en faveur de notre définition de la simulation, qui utilise une équivalence par dilatation temporelle.

En le généralisant pour tout $d \in \mathbb{N}^\star$ et pas seulement $d = 2$, cela serait l'équivalent d'un théorème d'accélération linéaire (speed-up) mais pour les ASMs au lieu des machines de Turing, c'est-à-dire pour les algorithmes séquentiels au lieu des fonctions calculables.

En particulier, cela permettrait d'ignorer les coefficients dans les classes en temps [1] et donc justifie pleinement l'usage des notations de Landau.

1. ↑ Ce que nous avons implicitement fait pour notre définition de la profondeur p.81.

Le modèle « augmenté »

Nous avons donc obtenu un modèle $\texttt{Imp}^+$ (voir la table B.7 p.155) capable de simuler les ASMs avec $d = 1$ et $e = 0$.

TABLE B.7 – Sémantique opérationnelle de $\texttt{Imp}^+$

$$\succ_1 \quad \begin{array}{r} \{(f_1\vec{t_1}, \ldots, f_m\vec{t_m}) := (t_1, \ldots, t_m); s\} \star X \\ \{s\} \star X \oplus \{(f_1, \vec{t_1}^X, \overline{t_1}^X), \ldots, (f_m, \vec{t_m}^X, \overline{t_m}^X)\} \end{array}$$

$$\succ_0 \quad \begin{array}{r} \{\texttt{if } F \ \{s_1\} \texttt{ else } \{s_2\}; s_3\} \star X \\ \{s_1; s_3\} \star X \text{ si } F \text{ est vraie dans } X \end{array}$$

$$\succ_0 \quad \begin{array}{r} \{\texttt{if } F \ \{s_1\} \texttt{ else } \{s_2\}; s_3\} \star X \\ \{s_2; s_3\} \star X \text{ si } F \text{ est fausse dans } X \end{array}$$

$$\succ_0 \quad \begin{array}{r} \{\texttt{while } F \ \{s_1\}; s_2\} \star X \\ \{s_1; \texttt{while } F \ \{s_1\}; s_2\} \star X \text{ si } F \text{ est vraie dans } X \end{array}$$

$$\succ_0 \quad \begin{array}{r} \{\texttt{while } F \ \{s_1\}; s_2\} \star X \\ \{s_2\} \star X \text{ si } F \text{ est fausse dans } X \end{array}$$

$$\succ_0 \quad \begin{array}{r} \{\texttt{loop } n \texttt{ except } F \ \{s_1\}; s_2\} \star X \\ \{s_1; \texttt{loop } n \texttt{ except } F \ \{s_1\}; s_2\} \star X \text{ si } i < n \text{ et } F \text{ est fausse dans } X \end{array}$$

$$\succ_0 \quad \begin{array}{r} \{\texttt{loop } n \texttt{ except } F \ \{s_1\}; s_2\} \star X \\ \{s_2\} \star X \text{ si } i = n \text{ ou } F \text{ est vraie dans } X \end{array}$$

Pour justifier $\texttt{Imp}^+ \simeq \texttt{ASM}$ il faudrait au moins donner l'idée de la réciproque.
Un m-uplet de mises à jour :

$$(f_1\vec{t_1}, \ldots, f_m\vec{t_m}) := (t_1, \ldots, t_m)$$

peut être immédiatement simulé par une parallélisation de mises à jour :

$$\texttt{par } f_1\vec{t_1} := t_1 \| \ \ldots \ \| \ f_m\vec{t_m} := t_m \texttt{ endpar}$$

Donc ce sont les autres commandes qui posent problème.

Seules les mises à jours comptent donc il faudrait définir une ASM passant d'une mise à jour à l'autre sans s'arrêter sur les autres commandes. Cela revient à déterminer à l'avance tous les chemins que l'exécution peut suivre entre deux mises à jour.

L'idée consiste à prendre une variable b_i déterminant si une mise à jour u_i est faite, ainsi qu'une variable b_{end} marquant la fin du programme. Puis le reste du programme détermine comment passer d'un b_i au b_j suivant (un seul booléen est vrai dans chaque état).

Ainsi, nous montrons à la table B.8 p.156 comment un programme impératif comprenant de la séquentialisation, une conditionnelle ou des boucles imbriquées peut être traduit à la main en ASM [1].

1. ↑ Pour simplifier les notations, les commandes `par`...`endpar` ont été rendues implicites par l'indentation.

TABLE B.8 – Traduction à la main

```
u1
   while F1
      if F2
         u2
      else
         u3
      while F3
         u4

if b1 then u1
if b2 then u2
if b3 then u3
if b4 then u4
if (¬b1) ∧ F3) then
   b4 := true
   if b2 then b2 := false
   if b3 then b3 := false
else
   if F1
      if F2
         b2 := true
         if b1 then b1 := false
         if b3 then b3 := false
         if b4 then b4 := false
      else
         b3 := true
         if b1 then b1 := false
         if b2 then b2 := false
         if b4 then b4 := false
   else
      bend := true
```

Bien sûr un exemple n'est pas preuve, et il n'est là que pour donner une idée de la traduction de $\texttt{Imp}^+$ dans `ASM`.

Dans la mesure où, par rapport au modèle classique, on ne peut que gagner du temps et non en perdre, les bornes de complexité en temps sont les mêmes. Ainsi, les mêmes fragments impératifs peuvent être utilisés pour caractériser les classes algorithmiques quelconques, primitives récursives et polynomiales.

De plus, en utilisant notre théorème p.103 nous aurions donc $\texttt{Imp}^+ \simeq \texttt{Imp}$, ce qui montrerait que selon notre définition de la simulation utiliser des m-uplets de mises à jour et rendre gratuites les conditionnelles et boucles ne devrait pas changer pas la classe algorithmique du langage.

Bibliographie

[APV10] P. Andary, B. Patrou & P. Valarcher – « A theorem of representation for primitive recursive algorithms », *Fundamenta Informaticae* **XX** (2010), p. 1–18. 10, 33, 44, 64, 74, 85, 109

[Ars91] J. Arsac – « Algorithmique et langages de programmation », *Bulletin de l'EPI* (1991), no. 64, p. 115–124. 9

[Bö05] E. Börger – « Abstract state machines : A unifying view of models of computation and of system design frameworks », *Annals of Pure and Applied Logic* (2005). 59

[BACS12] A. M. Ben-Amram, N. H. Christensen & J. G. Simonsen – « Computational models with no linear speedup », *Chicago Journal of Theoretical Computer Science* (2012), no. 7, p. 1–24. 107

[BBD+04] T. Biedl, J. F. Buss, E. D. Demaine, M. L. Demaine, M. Hajiaghayi & T. Vinar – « Palindrome recognition using a multidimensional tape », *Theoretical Computer Science* **302** (2004), p. 475–480. 8

[BC92] S. Bellantoni & S. Cook – « A new recursion-theoretic characterization of the polytime functions », *Computational Complexity* **2** (1992), p. 97–110. 80

[BDG09] A. Blass, N. Dershowitz & Y. Gurevich – « When are two algorithms the same ? », *Bullettin of Symbolic Logic* **15** (2009), p. 145–168. 9, 20, 23, 52

[BG02] A. Blass & Y. Gurevich – « Algorithms vs. machines », *Bullettin of the European Association for Theoretical Computer Science* (2002), no. 77, p. 96–118. 9, 111

[BG04] — , « Why sets ? », *Bullettin of the European Association for Theoretical Computer Science* (2004), no. 84. 5

[BG08] — , « Abstract state machines capture parallel algorithms », *ACM transactions on Computational Logic* **9** (2008). 10

[BGM10] P. Baillot, M. Gaboardi & V. Mogbil – « A polytime functional language from light linear logic », *Programming Languages and Systems* **6012** (2010), p. 104–124. 113

[CG10] P. Cégielski & I. Guessarian – « Normalization of some extended abstract state machines », *Fields of Logic and Computation, Lecture Notes in Computer Science* **6300** (2010), p. 165–180. 9

[CL03a] R. Cori & D. Lascar – *Logique mathématique – calcul propositionnel, algèbre de boole, calcul des prédicats*, Masson éd., vol. 1, Dunod, 2003. 15, 21, 117

[CL03b] — , *Logique mathématique – fonctions récursives, théorème de Gödel, théorie des ensembles, théorie des modèles*, Masson éd., vol. 2, Dunod, 2003. 7, 33, 44, 57, 78, 94

[Col91] L. Colson – « About primitive recursive algorithms », *Theoretical Computer Science* **83** (1991), p. 57–69. 6, 24, 35, 44

[Dav01] R. David – « On the asymptotic behaviour of primitive recursive algorithms », *Theoretical Computer Science* **266** (2001), p. 159–193. 7

[FZG10] M. Ferbus-Zanda & S. Grigorieff – « Asm and operational algorithmic completeness of lambda calculus », *Fields of Logic and Computation* (2010). 35, 89

[GGV01] U. Glaesser, Y. Gurevich & M. Veanes – « Universal plug and play machine models », *Technical report MSR-TR-2001-59 Microsoft Research* (2001). 22

[Gla67] M. D. Gladstone – « A reduction of the recursion scheme », *The Journal of Symbolic Logic* **32** (1967), no. 4. 7

[GS97] Y. Gurevich & M. Spielmann – « Recursive abstract state machines », *Journal of Universal Computer Science* (1997), no. 3, p. 233–246. 111

[Gur00] Y. Gurevich – « Sequential abstract state machines capture sequential algorithms », *ACM Transactions on Computational Logic* (2000). 9, 16, 22, 25, 27

[Gur05] — , « Interactive algorithms », *Mathematical Foundations of Computer Science* (2005). 22

[GV10] S. Grigorieff & P. Valarcher – « Evolving multialgebras unify all usual sequential computation models », *Symposium on Theoretical Aspects of Computer Science* (2010), p. 417–428. 73, 107, 139, 141

[GV12] — , « Classes of algorithms : Formalization and comparison », *Bulletin of the EATCS* (2012), no. 107. 60

[Jon99] N. D. Jones – « LOGSPACE and PTIME characterized by programming languages », *Theoretical Computer Science* (1999), no. 228, p. 151–174. 10, 45, 47

[Knu74] D. E. Knuth – « Surreal numbers : How two ex-students turned on to pure mathematics and found total happiness : A mathematical novelette », *Addison-Wesley Professional* (1974). 5

[Kri07] J.-L. Krivine – « A call-by-name lambda-calculus machine », *Higher Order and Symbolic Computation* (2007), no. 20, p. 199–207. 27

[Mar15a] Y. Marquer – « Algorithmic completeness of imperative programming languages », *Fundamenta Informaticae* (2015), En revue. 11

[Mar15b] — , « Imperative characterization of polynomial time algorithms », *Developments in Implicit Computational Complexity* (2015), Conférence. 11

[Mos01] Y. N. Moschovakis – « What is an algorithm ? », *Mathematics Unlimited* (2001). 9, 110

[Mos03] — , « On primitive recursive algorithms and the greatest common divisor function », *Theoretical Computer Science* **301** (2003), p. 1–30. 9

[MR67] A. Meyer & D. Ritchie – « The complexity of loop programs », *ACM national conference* (1967), no. 22, p. 465–469. 10, 43, 64, 74, 92

[MV09] D. Michel & P. Valarcher – « A total functional programming language that computes primitive recursive algorithms », (2009). 7, 10, 36, 44, 97, 109, 148

[Nee03] P. M. Neergaard – « Ploop : A language for polynomial time », (2003). 80, 113

[Sha92] A. Shamir – « Ip = pspace », *Journal of the ACM* **39** (1992), p. 869–877. 112

[Soa96] R. I. Soare – « Computability and recursion », *Bulletin of Symbolic Logic* (1996), no. 2, p. 284–321. 112

[Tur37] A. M. Turing – « On computable numbers, with an application to the entscheidungsproblem », *Proc. London Math. Soc.* **42** (1937), p. 230–265. 6

[Yan10] N. S. Yanofsky – « Galois theory of algorithms », (2010). 9

[Yan11] — , « Towards a definition of an algorithm », *Journal of Logic and Computation* **21** (2011), p. 253–286. 9

Index

A

Abstract State Machine 41
Accélération linéaire 154
`Algo` 22
$\texttt{Algo}_{\mathcal{C}}$ 33
Algorithmes
  identiques 23
  séquentiels 22
Arité 16
`ASM` 41

B

$\mathbb{B}$ 17
Booléens 17
Bornes d'une boucle 80
Bound 80

C

Commutation de contexte 48
Complexité
  en espace 74
  en temps 33
Composition de programmes 48
Compteur d'une boucle 46
Condition d'arrêt 94
Constructeur 30

D

Degré de parallélisme 89
depth 81
Dilatation temporelle 55

E

Écrasement 52
Éléments critiques 27
Entiers 144
  unaires 19, 143
Équivalence algorithmique 56
Exécution 23

F

Fonction
  monotone 74
  polynomiale 33
  primitive récursive 33
Formule 19
  garde 40
  sous forme normale conjontive ... 41
F_{Π} 94

G

Graphe 146
  d'exécution 62

I

Imbrication d'un programme 95
`Imp` 46
$\texttt{Imp}^{+}$ 155
Indentation 43
Insertion de programme 98
Interprétation
  d'un symbole 16
  d'un terme 19
  uniforme 29
Isomorphisme 20

L

Langage
  égalitaire 18
  du premier ordre 16
length 63
Liste 144
`Loop` 43
`LoopC` 46
$\texttt{LoopC}_{\texttt{stat}}$ 81

M

Matelassage 89, 98, 150
Mises à jour 25
  consistantes 26
  d'un programme impératif 51
  n-uplet 147
  triviales 26
$\ominus$ 26

N

$\mathbb{N}_1$ 19

nest 95
Nombre de Bell 40, 41, 148

O

Opération 30

P

$P(X)$ 47
P_{c_Π} 92
`PLoop` 80
$\oplus$ 26
$\mathcal{P}ol$ 33
Postulat
  premier 22
  second 24
  troisième 27
P_Π 89
$\mathcal{PR}$ 33
Profondeur d'un programme 81
Programme
  coquille 92, 97
  d'une ASM 36
  impératif 46
  noyau 89, 98
  sous forme normale 40

R

Read(Π) 37
Représentation 19, 142
Restriction de structure 54

S

Sémantique opérationnelle
  des ASMs 37
  des programmes impératifs 47
Simulation 56
  pas-à-pas 55
`skip` 89, 100
Speed-up 154
Stable
  par sous-termes 18
Structure
  du premier ordre 16
  mesurable 30
  représentable 30
Substitution 86, 154
Symbole
  de fonction 16
  de relation 18
  dynamique 25
  initial 30
  statique 25

T

Témoin d'exploration 27
Tableau 31, 145
Taille
  d'un état 32
  d'un symbole 31
  d'un terme 142
$\tau_P^i(X)$ 51
$\tau_\Pi^i(X)$ 37
Terminaison
  d'un algorithme 23
  d'un programme impératif 47
time 23, 47
t_Π 89
Type 141
  usuel 143

U

undef 16

V

Variables 15
  temporaires 53

W

`While` 46

Résumé / Summary

Les moyens sont la fin. — (Ursula K. Le Guin, *Les Dépossédés*)

Caractérisation impérative des algorithmes séquentiels en temps quelconque, primitif récursif ou polynomial

Les fonctions calculables ont été formalisées par différents modèles de calcul (récursion, lambda-calcul, machines de Turing) ayant le même comportement entrées-sortie : c'est la thèse de Church. Par exemple, une machine de Turing à un ruban peut simuler les résultats calculés par une machine à deux rubans. Pourtant, pour la reconnaissance de palindrome la machine à un seul ruban nécessite une complexité en temps supérieure. Ainsi, il faut étudier les étapes intermédiaires. Nous définissons donc une simulation pas à pas entre exécutions, utilisant uniquement des variables temporaires et une dilatation temporelle.

La thèse de Church concernait donc les fonctions, et il nous faut une nouvelle thèse pour les algorithmes. Nous avons choisi celle de Gurevich pour sa présentation axiomatique : un algorithme séquentiel est un objet vérifiant les trois postulats de temps séquentiel, d'états abstraits et d'exploration bornée. De plus, il a montré que les algorithmes séquentiels coïncident avec son modèle des Abstract State Machines. Nous dirons donc qu'un modèle de calcul est algorithmiquement complet s'il existe une simulation mutuelle d'exécutions entre lui et les ASMs.

Pour obtenir des résultats sur des modèles de calcul plus usuels, nous formalisons les langages impératifs par un système de transition. En étudiant leur sémantique opérationnelle pas à pas et en développant une notion de graphe d'exécution, nous montrons qu'une variante du langage `While` de Jones est algorithmiquement complète. Ce résultat étant à structures de données près, il correspond donc à une équivalence entre les structures de contrôle des ASMs et des langages impératifs. De plus, étant préoccupés par la faisabilité des algorithmes, nous montrons que les structures du premier ordre utilisées par Gurevich permettent bien d'implémenter les structures de données usuelles.

Notre soucis de la faisabilité nous a également conduit à étudier en terme de complexité implicite deux restrictions des ASMs : celles en temps primitif récursif (les opérations usuelles et terminales) et celles en temps polynomial (les exécutions réalistes). Nous montrons d'une part que si les structures de données sont également primitives récursives, alors une variante `LoopC` du langage `Loop` de Meyer et Ritchie est complète pour les algorithmes en temps primitif récursif. D'autre part, une restriction syntaxique $\texttt{LoopC}_{\texttt{stat}}$ de `LoopC` est complète pour les algorithmes en temps polynomial, sans restriction sur les structures de données et en caractérisant syntaxiquement le degré du polynôme.

Mots-clés. Algorithme séquentiel, ASM, calculabilité, complexité implicite, langage impératif, simulation, temps récursif primitif, temps polynomial.

Imperative Characterization of Sequential Algorithms in general, primitive recursive or polynomial time

Calculable functions were formalized by different computation models (recursion, lambda calculus, Turing machines) with the same input-output behavior : this is the Church's Thesis. For example, a one-tape Turing machine can simulate the results computed by a two-tapes machine. However, for the palindrome recognition the one-tape machine requires a greater complexity in time. Thus, we must study the intermediate steps. So we define a step-by-step simulation between executions, using only temporary variables and time dilation.

Church's thesis therefore concerns functions, and we need a new thesis for algorithms. We chose Gurevich's thesis for its axiomatic presentation : a sequential algorithm is an object satisfying the three postulates of sequential time, abstract states and bounded exploration. Moreover, he proved that the sequential algorithms coincide with his model of Abstract State Machines. So we will say that a computation model is algorithmically complete if there exists a mutual simulation of executions between it and the ASMs.

To obtain results on more common computation models, we formalize imperative programming languages by a transition system. By studying their operational semantics step by step and developing a notion of graph of execution, we prove that a variant of Jones' language `While` is algorithmically complete. This result is up to data structures, so it corresponds to an equivalence between the control flow statements of the ASMs and the imperative languages. Moreover, being concerned for the feasibility of algorithms, we prove that the first order structures used by Gurevich enable the implementation of usual data structures.

Our concern for feasibility has also led us to study in terms of implicit complexity two restrictions of the ASMs : those in primitive recursive time (the usual and terminal operations) and those in polynomial time (the realistic executions). Firstly, we prove that if the data structures are also primitive recursive, then a variant `LoopC` of Meyer and Ritchie's language is complete for the algorithms in primitive recursive time. Secondly, we prove that a syntactical restriction $\texttt{LoopC}_{\texttt{stat}}$ of `LoopC` is complete for the algorithms in polynomial time, without restriction on the data structures and with a syntactical characterization of the degree of the polynomial.

Keywords. ASM, computability, implicit complexity, imperative language, polynomial time, primitive recursive time, sequential algorithm, simulation.

Laboratoire d'Algorithmique, Complexité et Logique
Département d'Informatique
Faculté des Sciences et Technologie
Université Paris-Est Créteil Val de Marne
61 avenue du Général de Gaulle
94010 Créteil Cedex, France