Université Paris 7 - Denis Diderot — UFR d'informatique

Sur la notion d'observation en sémantique

Doctorat
Informatique

Benjamin Leperchey

Thèse dirigée par Antonio Bucciarelli

Jury

M. Nick Benton	
M. Gérard Boudol	
M. Antonio Bucciarelli	Directeur de thèse
M. Pierre-Louis Curien	
M. Jean Goubault-Larrecq	Rapporteur
M. John Longley	Rapporteur

Table des matières

1 Préliminaires : les λ-calculs typés et leurs modèles 19
1.1 Le λ-calcul simplement typé et PCF 19
1.1.1 λ-calcul simplement typé 19
1.1.2 PCF, PCF finitaire, PCF unaire 23
1.2 Modèle d'un λ-calcul étendu 25
1.2.1 Catégories cartésiennes fermées 25
1.2.2 Catégories CPO-enrichies 27
1.2.3 Modèles et modèles standard 28
1.2.4 Correction et adéquation 32
1.3 Modèles usuels 33
1.3.1 Domaines de Scott 34
1.3.2 Stabilité et cohérence 35
1.3.3 Stabilité forte 37
1.3.4 Jeux de Hyland et Ong 38
1.4 Monades 41
1.5 Équivalences contextuelles 43
1.5.1 Définitions 43
1.5.2 Cas du modèle de jeux 48
1.6 Relations logiques 51
1.6.1 Définition et propriétés 51
1.6.2 Relations de Sieber 52
1.6.3 Quotients 53

2 Définissabilité relative et hiérarchies de modèles 57
2.1 Généralités 57
2.1.1 Définissabilité relative 57
2.1.2 Hiérarchies de modèles 59
2.2 Définissabilité dans le modèle de Scott de PCF unaire 63
2.2.1 Quelques degrés de définissabilité 64
2.2.2 Hiérarchie des modèles standard 68

2.3 Définissabilité dans le modèle de Scott de PCF finitaire 71
2.3.1 Hypergraphes et h-morphismes 71
2.3.2 Morphismes temporisés 73
2.3.3 Fonctions sous-séquentielles 74
2.3.4 Correction 76
2.3.5 Complétude pour les fonctions sous-séquentielles 78
2.3.6 Complexité 80
2.3.7 Conclusions 82
2.4 Définissabilité dans les modèles stables 82
2.4.1 Arbres de décision 84
2.4.2 quote 85
2.4.3 La fonctionnelle H 88
2.4.4 Conclusions 89

3 Allocation dynamique 91
3.1 Le langage 93
3.1.1 Types et syntaxe 93
3.1.2 Sémantique opérationnelle 96
3.2 Modèle de base 100
3.2.1 FM-cpos 100
3.2.2 Définition de la sémantique 101
3.2.3 Correction et adéquation 105
3.2.4 Égalités dans le modèle, incomplétude 115
3.2.5 Égalités 115
3.2.6 Incomplétude 117
3.3 Relations paramétriques 118
3.3.1 Relations finitaires entre états et paramètres 119
3.3.2 Paramètres et relations paramétriques 123
3.3.3 Typage des états généralisé 132
3.4 Exemples 133
3.4.1 La suite de Meyer-Sieber 133
3.4.2 Typage généralisé 136
3.4.3 Memoisation 137
3.4.4 Réversibilité des changements (snapback) 137
3.4.5 Protocoles cryptographiques 138
3.5 Conclusion 140

4 Coût en sémantique dénotationnelle 143
4.1 Utilisation de ressources 143
4.2 PCF temporisé 145
4.2.1 Machine 145

4.2.2 PCF+tick 151
4.2.3 Modèle 153
4.3 Modèles temporisés 162
4.3.1 Modèle d'espaces de cohérence 162
4.3.2 Modèle de jeux 164
4.4 Conclusion 175

5 Conclusions 177

Table des figures

1.1 Règles de typage pour $\Lambda_{\mathcal{C}}$ 21
1.2 Sémantique opérationnelle pour $\Lambda_{\mathcal{C}}$ 22

3.1 Règles de typage 95
3.2 Sémantique opérationnelle 98
3.3 Sémantique opérationnelle (suite) 99
3.4 Sémantique des types 102
3.5 Sémantique des termes (extrait) 104
3.6 Égalités du métalangage 115
3.7 Quelques équations à propos de références 116
3.8 Définition des relations paramétriques 125

4.1 Règles de réduction de la machine en appel par nom. 147
4.2 Typage de la machine 148
4.3 Dénotations des termes 154
4.4 Dénotations des continuations 155
4.5 Exemple d'interaction 166

Introduction

Méthodes formelles en programmation

Ce qui différencie les ordinateurs des machines à calculer plus primitives, c'est la possibilité de les programmer. Les premières machines à calculer, par exemple celle de Pascal ou de Leibnitz, fonctionnaient de façon complètement mécanique : chaque opération devait être actionnée par une commande de l'utilisateur. Les calculs pouvaient être très complexes, et assez longs (on savait par exemple extraire des racines carrées), mais aucune automatisation n'était possible (hormis, bien sûr, en modifiant la machine). Babbage eut l'idée d'ajouter des calculs logiques aux opérations arithmétiques : sa machine savait prendre des décisions automatiquement en fonction des résultats. Cette idée a rendu possible la programmation d'une machine : on peut lui expliquer ce qu'elle doit faire, en lui donnant, en une seule fois, une suite de commandes indiquant ce qu'il faut calculer et comment prendre les décisions nécessaires. Le support physique et la forme de ces commandes ont changé, mais l'idée est restée la même : il s'agit de moyens d'expliquer à l'ordinateur comment exécuter un algorithme, c'est-à-dire une suite de calculs qui dépend de données et de résultats intermédiaires.

Aujourd'hui, toutes (ou presque) les machines électroniques sont programmables ou programmées : même pour une application très simple comme une montre à cristaux liquides, il revient souvent beaucoup moins cher d'utiliser un petit processeur déjà existant et de le programmer pour contrôler les différentes partie de la machine (boutons, écran etc.) que de construire une machine dédiée de A à Z. La seule différence avec un véritable ordinateur est que l'utilisateur n'a pas accès à ce programme et ne peut pas le modifier : les montres à cristaux liquides sauraient calculer π, mais le constructeur ne fournit qu'un programme qui affiche l'heure. De même, fours à micro-ondes, lecteurs de DVD, ascenseurs et avions contiennent des processeurs et des programmes : l'informatique et la programmation ont pris une importance capitale, et les "ordinateurs" n'en sont que la partie visible.

Cette évolution apporte beaucoup d'avantages. Au delà des coûts, réduits parce qu'il est beaucoup plus facile d'écrire un programme que de concevoir et produire du matériel, on gagne une souplesse incomparable : le programme, qui réside souvent dans une mémoire réinscriptible, peut être changé, pour corriger des erreurs ou ajouter de nouvelles fonctions. Ainsi, les derniers modems ADSL grand public (FreeBox, LiveBox) contiennent un système d'exploitation complet (à savoir Linux), ce qui leur permet d'effectuer des opérations complexes (de servir de routeur, de pare-feu etc.), et elles sont capables de télécharger automatiquement les mises à jour de ce logiciel.

Au contraire, la complexité croissante des applications embarquées rend leur conception et leur vérification de plus en plus ardues : s'il est assez facile de se convaincre, avant de la mettre sur le marché, qu'une montre fonctionne correctement, il est beaucoup plus difficile de tester toutes les combinaisons rendues possibles par un véritable système d'exploitation. On peut, et c'est l'approche la plus couramment utilisée, faire quelques tests, puis vendre le programme quand il a l'air de fonctionner correctement, en distribuant des mises à jour pour corriger les erreurs découvertes après-coup. Ceci pose plusieurs problèmes : d'abord, il faut être en mesure de remplacer le programme dans la machine : ce qui est très facile sur un modem, naturellement connecté à un réseau, est impossible sur une montre ou un four. Deuxièmement, les erreurs qu'on a pu laisser passer lors de la phase de tests initiale peuvent se révéler dangereuses : si, pour un modem, cela peut se traduire par une prise de contrôle des ordinateurs par un pirate, ce qui est déjà assez grave, des erreurs logicielles peuvent amener une fusée à exploser (cf. Ariane 5 en 1996), ou un avion à ne plus répondre aux commandes, avec des conséquences dramatiques.

Même sur les machines plus "traditionnelles" – des ordinateurs de bureau – la qualité du logiciel est de plus en plus importante : avec la mise en réseau, chaque erreur est une porte d'entrée potentielle pour des pirates. Au delà des programmes résidents sur la machine, les ordinateurs exécutent de plus en plus d'applications venant de l'internet : applets Java, Javascript etc. Les problèmes de sécurité potentiels sont énormes : il faut des moyens de contingenter ces programmes, pour les empêcher, par exemple, d'effacer le disque dur ou d'obtenir des mots de passe.

Ces deux problèmes – qualité des logiciels intégrés et prudence vis-à-vis des logiciels "inconnus" – montrent qu'il devient indispensable de formaliser la programmation : en décrivant mathématiquement le comportement d'un programme, ou plutôt tous les comportements possibles d'un programme, on pourra prouver qu'il vérifie certaines propriétés : que le programme fait ce qu'on attend de lui, qu'il répond dans un délai raisonnable, qu'il ne peut

pas appeler (ou faire appeler) des fonctions dangereuses. Ces propriétés sont toutes indécidables. En pratique, on utilisera un assistant de preuve pour vérifier que les preuves sont correctes et qu'on n'a pas oublié de cas, ou bien on se contentera de vérifier des propriétés plus basiques (typage, mémoire bornée, indices de tableaux etc.), qui sont décidables et suffisent à éliminer des classes entières d'erreurs.

Ces vérifications pourront se faire à deux niveaux : sur les programmes locaux, on connaît le texte original qui représente la volonté du programmeur au delà des détails (allocation de registres etc.) et c'est à ce niveau qu'on pourra espérer prouver des propriétés avancées. Au contraire, les éditeurs qui distribuent des programmes sur internet les donnent le plus souvent sous une forme compilée (en bytecode Java par exemple). On ne pourra en pratique vérifier que des propriétés plus simples (comme le bon typage de la pile), et les autres vérifications (droit d'appeler telle ou telle fonction) seront dynamiques, c'est-à-dire lors de l'exécution du programme.

Cette thèse ne s'intéresse qu'au premier cas : il s'agira de modéliser des langages de programmation de haut niveau. Ce domaine de recherche a été initié par Scott et Strachey [SS71]. En fait, on ne décrira pas un langage de programmation courant (comme C, Java, ou Ada), mais un petit langage, PCF, utilisé depuis les années 70 pour essayer des outils mathématiques qu'on pourra ensuite mettre en oeuvre sur des langages plus réalistes. Étant fonctionnel, il est plus proche de ML [MTHM97], ou même de Haskell [PHA+97], puisqu'il utilise l'appel par nom. Il a plusieurs avantages théoriques :

- il est très régulier : sa syntaxe est définie par une grammaire hors-contexte et sa sémantique opérationnelle est définie par la structure du terme considéré, avec un simple prédicat $P \Downarrow Q$,
- il est extrêmement simple : un seul type de base (les entiers), et quatre constructions (variable, fonction récursive, application et branchement conditionnel),
- il repose sur des concepts mathématiques bien compris (les fonctions), ce qui donne une intuition claire de sa sémantique.

Évidemment, il n'est pas du tout réaliste :

- Les langages courants sont déclaratifs et pas fonctionnels : ils utilisent des suites d'instructions plutôt que des applications de fonctions. La plupart des résultats seront difficiles à traduire pour les langages usuels.
- Leurs algèbres de types sont beaucoup plus riches : flottants, entiers sur 32 bits, objets avec méthodes etc. Les encodages dans PCF sont possibles, mais ils ne sont ni pratiques ni précis.
- Ils offrent des possibilités plus vastes, en particulier des sauts (goto), des exceptions, des variables modifiables (ou références).

On pourrait toutefois avancer que le premier argument constitue une bonne

raison de préférer les langages fonctionnels aux langages impératifs quand la qualité du logiciel est cruciale : la possiblité d'utiliser des outils mathématiques précis contrebalance alors largement les pertes de performances, ou les coûts de développement (bien que beaucoup de programmes soient plus facile à écrire dans les langages fonctionnels, les programmeurs n'y connaissent souvent rien). À ce sujet, on pourra relire [Hug89].

Le deuxième problème devrait être assez facile à résoudre, car la théorie est souvent indépendante des domaines de base. Toutefois, il n'y a pas encore de consensus dans la communauté des langages fonctionnels sur la programmation orientée objets.

Le troisième était a priori le plus difficile, mais on sait aujourd'hui modéliser les opérateurs de contrôle "bien élevés" (en particuliers, les exceptions), et les références, en utilisant des monades, ou directement, par exemple en sémantique des jeux. Encore une fois, on pourrait avancer que les constructions qui ne rentrent pas dans ces cadres théoriques sont trop ésotériques ou trop dangereuses (goto) pour être intégrées à un langage de programmation courant.

Il reste tout de même un fossé entre la pratique de la programmation et les développements théoriques. À part dans les transports (trains et avions), ou les méthodes formelles deviennent la norme, parce que les erreurs sont interdites, l'industrie du logiciel utilise à peine les possibilités offertes par la sémantique des langages de programmation. On peut cependant espérer que, les problèmes devenant à la fois plus nombreux et plus importants, la qualité deviendra le principal atout de compétitivité dans l'industrie du logiciel, devant le prix.

Sémantique d'un langage de programmation

Qu'est-ce qu'un langage de programmation ?

Pour mieux comprendre ce que veut dire programmer et quelle forme prennent les programmes, il faut comprendre que les machines d'aujourd'hui ne font plus que de la logique. Les chiffres 0 et 1, appelés bits (pour binary unit), suffisent à représenter n'importe quel nombre entier : c'est l'écriture binaire. Ainsi, 0 s'écrit "0", 1 s'écrit "1", 2 s'écrit "10", 27 s'écrit "11011", etc. On a donc réutilisé les valeurs de vérité "vrai" et "faux" pour coder 0 et 1 : la logique, initialement cantonnée à la partie "programme" des machines mécaniques, s'est également imposée dans les parties calculatoires. Tous les nombres sont représentés comme des suites de bits, et les opérations sur les nombres comme des "raisonnements" mécaniques. Le programme consistant

à organiser les différentes phases de calculs à l'aide de tests et de conditions de vérités ("si le résultat est 0, alors arrête"), il est très naturellement écrit en binaire : tableaux d'interrupteurs ouverts ou fermés, ou, dans le cas des cartes perforées, absence ou présence de trous. Un programme est donc une suite de nombres binaires, un pour chaque instruction. Ce "langage" rudimentaire s'appelle le langage machine : c'est la façon naturelle (pour la machine !) de communiquer. Évidemment, programmer "à la main" devient très vite extrêmement pénible : les suites de chiffres codant les instructions ne sont en général pas très intuitives, et les erreurs sont très fréquentes.

On a donc cherché des moyens de rendre la programmation plus simple. Un premier pas consiste tout simplement à donner un nom aux instructions, et à automatiser la traduction de ces noms en codes. A partir de là, il est aussi facile d'automatiser le calcul des longueur des sauts dans le programme, en utilisant des étiquettes. Le langage obtenu (ou plutôt, les langages, puisqu'ils dépendent de la machine) s'appelle assembleur.

Ce processus étant très simple, il est facile d'écrire un programme qui le fait automatiquement. On a alors un programme A qui prend en entrée un programme écrit en assembleur (un texte), et qui produit un programme en langage machine (une suite de bits). Pour la machine, ce programme A est un programme comme les autres : on peut l'écrire pour et l'exécuter sur un ordinateur de bureau, un serveur internet ou une calculatrice programmable. Encore mieux, on peut l'écrire pour un type de machine sur une autre machine : il existe des traducteurs d'assembleur PowerPC en code machine PowerPC qui tournent sur les processeurs Intel et vice-versa.

Il reste néanmoins de nombreux problèmes :

- d'abord, les instructions restent élémentaires, ce qui rend le programme très long : il reste donc très difficile de ne pas faire d'erreurs,
- le programme n'est valable que sur un type de machine : un programme en assembleur écrit pour le processeur Intel d'un PC n'a aucun sens pour le PowerPC d'un Macintosh.

Ces deux problèmes sont symptômes d'un même mal : l'assembleur n'introduit aucune abstraction. Si on pouvait exprimer les commandes en termes plus généraux, les programmes seraient plus courts et plus faciles à écrire, et, si ces termes ne sont pas dédiés à un processeur particulier, il sera plus facile d'exécuter le programme sur différentes machines. Il faut donc définir un langage de plus haut niveau, qui abstrait les différences entre machines et automatise les petites séries d'instructions, comme celles qu'il faut placer au début et à la fin des appels de fonction. Chaque instruction du programme est donc traduite en une série d'instructions machine : ce processus s'appelle compilation. L'exemple-type d'un tel langage est le C : on donne des noms aux variables, ce qui améliore la lisibilité du programme et on permet les

expressions complexes comme $3x + 7$ (en assembleur, une instruction correspond en général à une seule opération). Les instructions restent assez simples pour que la compilation soit facile, mais sans aucune référence à un processeur particulier, ce qui permet d'écrire des compilateurs pour n'importe quel machine.

Interprétation, et sens d'un programme

Un problème se pose : comme la correspondance entre le programme en langage clair (en C) et le programme en langage machine n'est pas évidente, il faut faire confiance au compilateur. S'il contient une erreur, le programme produit peut être erroné alors que le programme d'origine était parfaitement correct. Même en écartant ce cas, les commandes de plus haut niveau ont un sens beaucoup moins clair, ce qui peut conduire à des erreurs du programmeur. En C, les instructions y = x++ ; et y = ++x ; sont similaires mais leur effet est très différent : dans les deux cas, la variable y prend la valeur de la variable x, qui est incrémentée, mais dans le premier cas, l'affectation est faite avant l'incrémentation, alors que dans l'autre, elle est faite après. Dans les deux cas, la valeur de x après exécution est la même, mais y contient, selon le cas, la valeur d'origine ou la valeur finale de x.

Le but d'un programme étant d'expliquer de manière non ambiguë à la machine ce qu'elle doit faire, il n'est pas envisageable de laisser des zones de flou dans le langage de programmation lui-même. Il faut donc, pour chaque instruction, définir précisément un comportement : c'est la sémantique du langage de programmation. Ainsi, pour notre exemple, on pourrait écrire les descriptions suivantes :

- var = expr calcule la valeur de expr, la met dans var et la renvoie,
- var++ incrémente var et renvoie son contenu initial,
- ++var incrémente var et renvoie son contenu final,

Ces règles permettent de comprendre la différence entre les instructions y = x++ et y = ++x. Ces descriptions, qui ont pourtant l'air très précises, cachent pourtant encore beaucoup d'ambiguïtés : par exemple, selon le type de la variable var, incrémenter aura un sens différent (dans presque tous les langages courants, les nombres entiers sont compris modulo $2^8, 2^{16}, 2^{32}$ ou 2^{64}). On voit donc l'avantage de définir une sémantique formelle, écrite en termes mathématiques qui ne laisse aucune ambiguïté.

On distingue habituellement deux classes de sémantiques : les sémantiques opérationnelles, qui manipulent des phrases du langage, et les sémantiques dénotationnelles, qui associent un objet mathématique à chaque phrase. La sémantique opérationnelle, en général plus facile à définir, est souvent plus proche de l'implémentation qu'on pourrait faire du langage, en tous cas elle

garde toujours un aspect concret, ce qui rend les preuves parfois très complexes. Au contraire, la sémantique dénotationnelle tend justement à abstraire le texte du programme pour mettre en valeur son sens, ce qui rend les preuves beaucoup plus simples puisqu'elles sont faites dans un monde mathématique ou il est facile de calculer. Le prix à payer est en général qu'on perd en précision lors du passage dans le monde abstrait.

Le plan d'étude d'un langage est en général le suivant :

1. On définit la syntaxe du langage i.e. les règles de constructions des phrases, en général par une grammaire hors-contexte.
2. On définit la sémantique opérationnelle de ce langage, qui servira de référence : elle décrit tous les comportements possibles du programme, et seulement ceux-ci.
3. On définit une sémantique dénotationnelle, et on prouve qu'elle correspond bien à la sémantique opérationnelle (correction et adéquation)
4. On utilise la sémantique dénotationnelle pour prouver des propriétés sur les termes du langage.

Plan de cette thèse

Cette thèse explore la notion d'observation : ce qu'on essaie de représenter par la sémantique, ce n'est pas le déroulement du programme dans une certaine implémentation du langage, mais son comportement global, ou encore son comportement visible, observable par l'extérieur.

La première partie rappelle des définitions et des notations pour des variantes de PCF, présente des modèles usuels, et définit l'équivalence observationnelle et la complète adéquation (full abstraction), c'est-à-dire l'équivalence entre observation dans le langage et observation dans le modèle.

Dans la deuxième partie, on examinera les rapports entre définissabilité et observation. On montrera, avec le cas de PCF unaire, que les hiérarchies de définissabilité et de modèles sont distinctes (il n'y a que deux modèles extensionnels et au moins quatre degrés de définissabilité). On examinera aussi le cas de PCF finitaire, en présentant les morphismes temporisés d'hypergraphes, qui sont un moyen de décider des degrés de définissabilité dans le modèle de Scott au premier ordre, complet pour les fonctions sous séquentielles. Alors que, dans les modèles de Scott, ces observations interdites sont en fait des évaluations en parallèle, on montrera, en reprenant les travaux de Longley, que les observations interdites dans le modèle fortement stable tiennent plutôt de la décompilation, ou du traçage des programmes.

La troisième partie est une étude de cas sur un langage plus ambitieux : on a ajouté des références. Ici, la notion d'observation par le contexte prend un sens un peu différent : il peut non seulement examiner les résultats du programme, mais aussi les changement de l'état de la machine. Il s'agira, à l'aide de relations logiques, de bien séparer la partie visible de la mémoire de la partie privée, accessible seulement par le programme : les contextes qui font des observations sur cette dernière seront éliminés. Cette nouvelle notion d'observation se rapproche de l'analyse de flot d'information pour le confidentialité : on donnera un exemple inspiré (de loin) des protocoles cryptographiques.

La dernière partie s'attache à un problème légèrement différent : au lieu d'étudier les différences entre observation syntaxique et observation dans le modèle pour essayer de les réduire, on augmente l'observation de la sémantique opérationnelle d'une information sur le coût du programme (qui n'est pas observable dans le langage). On montre comment instrumenter un modèle dénotationnel existant pour y représenter également le coût : plus précisément, on donne les axiomes que doit vérifier le modèle de départ, et la construction générique qui permet d'obtenir un modèle correct et adéquat. On donne deux exemples, dans les espaces de cohérence et dans les jeux. Le modèle de jeux est complètement adéquat pour la notion d'observation correspondant au coût (l'amélioration). On présente finalement une nouvelle classe de stratégies, les stratégies semi alternantes, qui définissent une notion de coût intrinsèque au modèle dénotationnel.

Chapitre 1

Préliminaires : les λ-calculs typés et leurs modèles

1.1 Le λ-calcul simplement typé et PCF

1.1.1 λ-calcul simplement typé

Le λ-calcul a été défini par Church vers 1930. Il s'agissait à l'origine d'une tentative de fondation des mathématiques. Il s'est avéré que cette théorie était incohérente : aujourd'hui, il est utilisé comme base théorique des langages de programmation dits fonctionnels.

On utilise surtout ses variantes typées, qui forment des théories cohérentes : cela signifie pour les langages de programmation correspondants (comme Caml [MTHM97] ou Haskell [PHA+97]) qu'un programme correct ne provoquera pas d'erreur lors de son exécution. Comme les opérateurs de récursion (θ etc.) du λ-calcul pur ne sont pas typés, on remplace la λ-abstraction classique par une construction plus générale pour définir les fonctions récursives. Le λ-calcul simplement typé n'est néanmoins pas très adapté pour l'étude des langages réels, car toutes les valeurs sont encodées à travers des fonctions : en particulier, les entiers sont le plus souvent représentés comme des itérateurs, ce qui n'est ni pratique ni réaliste. On va donc ajouter des constantes au λ-calcul simplement typé.

Dans toute la suite, $\mathcal{C}$ sera un ensemble de constantes.

Définition 1.1.1 (Types simples)

Étant donné un type de base ι, les types simples (notés $A, B \dots$) sont construits à partir du type de base avec la flèche :

$$A ::= \iota \mid A \to B$$

Définition 1.1.2 (Termes)

On utilise la notation de Krivine [Kri90]. On se donne un ensemble infini de variables $x, y \ldots$. $\mathcal{C}$ est un ensemble (fini ou infini) de constantes $c_1, c_2 \ldots$, disjoint des variables. Les termes du λ-calcul simplement typé sont :

$$\begin{array}{rcll} M & ::= & x, y \ldots & \text{Variables} \\ & | & \mathsf{rec}\, f\, (x : A) = M & \text{Abstraction/Récursion} \\ & | & (M)N & \text{Application} \\ & | & c_1, c_2 \ldots & \text{Constantes} \end{array}$$

L'ensemble des termes est noté $\Lambda_{\mathcal{C}}$. Une valeur est un terme qui n'est pas une application. Un terme est fermé s'il n'a pas de variable libre.

Pour les fonctions non récursives, on notera $\lambda(x : A).M = \mathsf{rec}\, f\, (x : A) = M$, où f n'est pas libre dans M. On omettra aussi souvent le type des variables liées.

Définition 1.1.3 (Substitution)

Si t, u sont dans $\Lambda_{\mathcal{C}}$ et x est une variable, on définit la substitution de x par N dans M, notée $M[x\backslash N]$ par induction sur M :

- $x[x\backslash N] = u$,
- $y[x\backslash N] = y$ si y est une variable distincte de x,
- $((M)N)[x\backslash N] = (M[x\backslash N])N[x\backslash N]$,
- $(\mathsf{rec}\, f\, x = M)[f\backslash N] = \mathsf{rec}\, f\, x = M$,
- $(\mathsf{rec}\, f\, x = M)[x\backslash N] = \mathsf{rec}\, f\, x = M$,
- $(\mathsf{rec}\, f\, x = M)[y\backslash N] = \mathsf{rec}\, f'\, x' = M[f\backslash f', x\backslash x'][y\backslash N]$, où f', x' ne sont pas libres dans M ni dans N et sont différents de y,
- $c[x\backslash N] = c$.

Définition 1.1.4 ($\boldsymbol{\alpha}$-conversion)

L'α-conversion est la plus petite congruence qui contient toutes les paires $\mathsf{rec}\, f\, x = M \equiv_\alpha \mathsf{rec}\, f'\, x' = M'$ où $M' = M[x\backslash x'][f\backslash f']$. Il s'agit simplement du renommage des variables liées.

On raisonnera toujours modulo α-conversion.

Typage structurel

On ne considérera que les termes typés : cela permet d'éliminer a priori une large classe d'erreur. Ainsi, on ne pourra pas exécuter un entier ou incrémenter une fonction.

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{}{\Gamma \vdash c : A_{\mathcal{C}}(c)}$$

$$\frac{f : A \to B, x : A, \Gamma \vdash M : B}{\Gamma \vdash \mathsf{rec}\ f\ (x : A) = M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M)N : B}$$

Fig. 1.1 – Règles de typage pour $\Lambda_{\mathcal{C}}$

Chaque constante c doit avoir un type (unique) qu'on note $A_{\mathcal{C}}(c)$, au plus du premier ordre (i.e. $A_{\mathcal{C}}(c)$ est ι ou $\iota \to \cdots \to \iota \to \iota$).

Un environnement de type Γ est une fonction partielle finie des variables dans les types simples. On dit que $\Gamma \subseteq \Gamma'$ si $\forall x, \forall A, \Gamma(x) = A \Rightarrow \Gamma'(x) = A$.

On présente les termes typés avec un ensemble de règles d'inférence : cela permet à la fois de décrire facilement la définition inductive du type d'un terme (s'il en a un), et rappelle fort à propos le calcul des séquents : les règles du λ-calcul typé dans sa version sans récursion et si on ne garde que les types (en omettant les termes), correspondent exactement à la logique intuitionniste minimale. Les règles de typage pour $\Lambda_{\mathcal{C}}$ sont données en figure 1.1.

Proposition 1.1.5

On suppose $\Gamma \vdash M : A$.
- à Γ, M donnés, il existe un unique A tel que $\Gamma \vdash M : A$ est dérivable, et la dérivation est unique,
- si $\Gamma \subseteq \Gamma'$, alors $\Gamma' \vdash M : A$,
- si $\Gamma = \Gamma', x : B$ et $\vdash N : B$, alors $\Gamma' \vdash M[x \backslash N] : A$.

Démonstration : L'unicité de la dérivation de typage découle de la présence explicite de types pour les variables liées. ■

Sémantique opérationnelle structurelle

On veut maintenant définir l'exécution d'un programme. Plusieurs choix s'offrent à nous : on pourrait la décrire pas à pas par une machine (sémantique à petits pas) ou bien décrire directement le résultat du programme (sémantique à grands pas). Nous avons choisi la sémantique à grands pas car elle combine plusieurs avantages :

- elle est plus compacte : on définit seulement un prédicat $\Downarrow$,

$$\frac{}{\mathsf{rec}\ f\ x = P \Downarrow \mathsf{rec}\ f\ x = P} \qquad \frac{}{c \Downarrow c}$$

$$\frac{P \Downarrow \mathsf{rec}\ f\ x = P' \qquad P'[x\backslash Q, f\backslash \mathsf{rec}\ f\ x = P'] \Downarrow V}{(P)Q \Downarrow V}$$

$$\frac{P \Downarrow c \qquad Q \Downarrow c'}{(P)Q \Downarrow \mathsf{eval}_\delta(c, c')}$$

Fig. 1.2 – Sémantique opérationnelle pour $\Lambda_{\mathcal{C}}$

– elle est définie par induction, plutôt que par une suite de réductions, ce qui rend souvent les preuves plus faciles.

Une sémantique à grands pas "écrase" toutes les étapes intermédiaires de calcul : c'est souvent souhaitable, mais on verra dans le chapitre 4 (où on utilisera une sémantique à petits pas) que ce n'est pas toujours préférable.

Évidemment, les constantes ne sont en général pas inertes : il faut donc, pour définir le comportement des programmes, définir le comportement des constantes :

Définition 1.1.6 (δ-règles)

On se donne la sémantique des constantes par des δ-règles, sous la forme d'une fonction partielle $\mathsf{eval}_\delta : \mathcal{C} \times \mathcal{C} \mapsto \mathcal{V}$, où $\mathcal{V}$ est l'ensemble des valeurs (constantes et abstractions), qui doit respecter le typage dans le sens que quand $\mathsf{eval}_\delta(c, c')$ est défini,

$$\mathsf{eval}_\delta(c, c') = V \quad \Rightarrow \quad \exists A, \begin{cases} A_{\mathcal{C}}(c) = \iota \to A \\ A_{\mathcal{C}}(c') = \iota \\ \vdash V : A \end{cases}$$

On rappelle que les constantes fonctionnelles s'appliquent à un entier, ce qui explique la forme de la condition. On peut maintenant définir le prédicat binaire $- \Downarrow -$ par les règles données en figure 1.2.

Proposition 1.1.7 (Préservation du typage (Subject reduction))

Si $\vdash P : A$ et $P \Downarrow Q$, alors $\vdash Q : A$.

Démonstration : Par induction sur la preuve de $P \Downarrow Q$. Si c'est un axiome, alors $P = Q$ et la conclusion est triviale.

On suppose $P \Downarrow \mathsf{rec} f x = P', P'[x\backslash Q, f\backslash \mathsf{rec} f x = P'] \Downarrow V$, et $\vdash (P)Q : A$. Il existe B tel que $\vdash P : B \to A$ et $\vdash Q : B$. Par hypothèse d'induction $\vdash \mathsf{rec}\ f\ x = P' : B \to A$, ce qui donne (par unicité de la dérivation de

typage), $x : B, f : B \to A \vdash P' : A$. On a donc

$$\vdash P'[x\backslash Q, f\backslash \mathsf{rec}\ f\ x = P'] : A$$

ce qui nous donne, encore par hypothèse d'induction, $\vdash V : A$.

Si $P \Downarrow c$, $Q \Downarrow c'$ et $\vdash (P)Q : A$, il existe B tel que $\vdash P : B \to A$ et $\vdash Q : B$. Par hypothèse d'induction $\vdash c : B \to A$ et $\vdash c' : B$, et par hypothèse sur eval_δ, $\vdash \mathsf{eval}_\delta(c, c') : A$. ■

1.1.2 PCF, PCF finitaire, PCF unaire

PCF

PCF est un langage de programmation simplifié, dérivé du système d'aide à la preuve LCF [GMW79], défini par Plotkin [Plo77] en 1977. Il est depuis devenu l'objet d'étude d'une grande partie des travaux en sémantique dénotationnelle. On ne reprendra pas exactement ici les définitions originelles : PCF est plus un genre qu'un langage bien défini. En particulier, nous avons regroupé l'abstraction $\lambda x.P$ et le point fixe YQ en une seule opération (la récursion $\mathsf{rec}\, f\, x = P$), ce qui rend la définition et les preuves plus compactes : on se convaincra facilement que les traductions suivantes montrent que les deux présentations sont équivalentes.

$$\begin{array}{rcll} \mathsf{rec}\ f\ x = P & \equiv_{\text{def}} & Y(\lambda f.\lambda x.P) & \\ \lambda x.P & \equiv_{\text{def}} & \mathsf{rec}\ f\ x = P & \text{où } f \text{ n'est pas libre dans } P \\ Y & \equiv_{\text{def}} & \mathsf{rec}\ f\ x = x(fx) & \end{array}$$

Définition 1.1.8 (PCF)

PCF est le λ-calcul simplement typé avec un type de base N pour les entiers, et des constantes :

- pour chaque $n \in \mathbb{N}$, $\underline{n}$ de type N,
- pred et succ de type $\mathsf{N} \to \mathsf{N}$,
- if de type $\mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}$.

Les δ-règles sont :

$$\begin{array}{l} \mathsf{eval}_\delta(\mathsf{succ}\ , \underline{n}) = \underline{n+1} \\ \mathsf{eval}_\delta(\mathsf{pred}\ , \underline{n+1}) = \underline{n} \\ \mathsf{eval}_\delta(\mathsf{if}, \underline{0}) = \lambda x.\lambda y.x \\ \mathsf{eval}_\delta(\mathsf{if}, \underline{n+1}) = \lambda x.\lambda y.y \end{array}$$

On notera $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P\ \mathsf{fi}$ plutôt que $\mathsf{if}\ MNP$.

Proposition 1.1.9

Un terme de PCF $M : \mathsf{N}$ converge si et seulement s'il existe un entier $n \in \mathbb{N}$ tel que $M \Downarrow \underline{n}$.

Démonstration : C'est une conséquence de la préservation du typage : les seuls termes de type N qui ne peuvent pas être réduits sont les constantes. ■

PCF finitaire

C'est le λ-calcul typé avec un nombre fini de constantes de base. Ce nombre précis n'est en fait pas important : on n'en utilisera que deux. Dans ce cas, comme dans le cas de PCF unaire qui suit, la présence de la récursion est un peu pédante : comme il n'y a qu'un nombre fini de comportements à chaque type, l'abstraction simple (non récursive) et une constante Ω (pour la non terminaison) donneraient un langage équivalent. Par souci d'uniformité de présentation, et parce que les programmes qui ne terminent pas sont en pratique des boucles infinies, nous avons gardé la récursion, et défini Ω.

Définition 1.1.10 (PCF finitaire)

PCF est le λ-calcul simplement typé avec un type de base B pour les booléens, et un test. Il y a deux constantes tt, ff de type B et une constante if de type $\mathsf{B} \to \mathsf{B} \to \mathsf{B} \to \mathsf{B}$. Les δ-règles sont :

$$\mathsf{eval}_\delta(\mathsf{if}, \mathrm{tt}) = \lambda x.\lambda y.x$$
$$\mathsf{eval}_\delta(\mathsf{if}, \mathrm{ff}) = \lambda x.\lambda y.y$$

On notera encore $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P\ \mathsf{fi}$ pour $\mathsf{if}\ MNP$ et

$$\Omega = (\mathsf{rec}\ f\ (x : \mathsf{B}) = (f)x)\mathrm{tt}$$

un terme de type B qui ne termine pas.

PCF unaire

PCF unaire est encore plus simple. On ne s'intéresse plus qu'à la terminaison des programmes :

Définition 1.1.11 (PCF unaire)

PCF unaire est le λ-calcul étendu par :
- un type de base U,
- une constante de type $\mathsf{U} : \top$,

– un opérateur binaire seq, de type $\mathsf{U} \to \mathsf{U} \to \mathsf{U}$.

La seule δ-règle est :

$$\mathsf{eval}_\delta(\mathsf{seq}, \top) = \lambda x.x$$

On notera $M; N$ au lieu de $((\mathsf{seq})M)N$. Encore une fois, on définit $\Omega = (\mathsf{rec}\ f\ (x : \mathsf{U}) = (f)x)\top$.

1.2 Modèle d'un λ-calcul étendu

Il existe de nombreuses définitions de modèle du lambda-calcul simplement typé. Nous utiliserons la définition catégorique (une catégorie cartésienne fermée), car elle est générale (en particulier, les domaines ne sont pas forcément des fonctions) et parce que nous utiliserons des constructions catégoriques (des monades) dans la dernière partie.

1.2.1 Catégories cartésiennes fermées

Par souci de complétude, on rappelle les définitions de base de théorie des catégories nécessaires pour la suite. Pour plus de détails, on pourra consulter [Mac71].

Définition 1.2.1 (Catégorie)

Un graphe orienté $\mathcal{G}$ est
- un ensemble d'objets, noté $\mathcal{C}_0$,
- un ensemble de morphismes, noté $\mathcal{C}_1$, avec des applications $s, d : \mathcal{C}_1 \to \mathcal{C}_0$ (source et destination)

Une catégorie $\mathcal{C}$ est un graphe orienté muni d'une opération de composition $\circ$ telle que
- $f \circ g$ existe si et seulement si $s(f) = d(g)$, et alors $d(f \circ g) = d(f)$ et $s(f \circ g) = s(g)$,
- $\circ$ est associative : si $s(f) = d(g)$ et $s(g) = d(h)$, $(f \circ g) \circ h = f \circ (g \circ h)$,
- pour chaque objet A, il existe une identité id_A pour $\circ$: si $s(f) = A$ et $d(f) = B$, $f \circ \mathsf{id}_A = f = \mathsf{id}_B \circ f$.

On notera $\mathcal{C}[A, B]$ les éléments f de $\mathcal{C}_1$ tels que $s(f) = A$ et $g(f) = B$.

Définition 1.2.2

Si $\mathcal{C}$ est une catégorie, $\mathcal{C}^{op}$ est la catégorie duale : les objets sont les mêmes et les flèches sont inversées. Si $\mathcal{C}, \mathcal{D}$ sont deux catégories, $\mathcal{C} \times \mathcal{D}$ est la catégorie produit :
- $(\mathcal{C} \times \mathcal{D})_0 = \mathcal{C}_0 \times \mathcal{D}_0$ et $(\mathcal{C} \times \mathcal{D})_1 = \mathcal{C}_1 \times \mathcal{D}_1$

– $s_{\mathcal{C}\times\mathcal{D}}(\langle f, g\rangle) = \langle s_{\mathcal{C}}(f), s_{\mathcal{D}}(g)\rangle$ et $d_{\mathcal{C}\times\mathcal{D}}(\langle f, g\rangle) = \langle d_{\mathcal{C}}(f), d_{\mathcal{D}}(g)\rangle$

Définition 1.2.3 (Foncteur)

Un foncteur F d'une catégorie $\mathcal{C}$ dans une catégorie $\mathcal{D}$ est une application de $\mathcal{C}_0$ dans $\mathcal{D}_0$ et une application de $\mathcal{C}_1$ dans $\mathcal{D}_1$ telles que
- $F(\mathsf{id}_A) = \mathsf{id}_{F(A)}$
- pour tout morphisme f de $\mathcal{C}$, $s(F(f)) = F(s(f))$ (resp. d)
- pour toute paire de morphismes composables f, g, $F(f \circ g) = F(f) \circ F(g)$.

Définition 1.2.4 (Hom-foncteurs)

Si $\mathcal{C}$ est une catégorie, on définit un bifoncteur $\mathcal{C}[-,-] : \mathcal{C}^{op} \times \mathcal{C} \to \mathsf{Set}$ par

$$\mathcal{C}[-,-](A, B) = \mathcal{C}[A, B]$$
$$\mathcal{C}[-,-](f, g) = \lambda h.g \circ h \circ f$$

Si A est un objet de $\mathcal{C}$, on définit $\mathcal{C}[A,-] : \mathcal{C} \to \mathsf{Set}$ et $\mathcal{C}[-, A] : \mathcal{C}^{op} \to \mathsf{Set}$ comme les foncteurs obtenus en contraignant la première ou la deuxième variable.

Définition 1.2.5 (Transformation naturelle)

Si F, G sont deux foncteurs de $\mathcal{C}$ dans $\mathcal{D}$, une transformation naturelle $\eta : F \Rightarrow G$ est une famille de morphismes $\eta_A : FA \to GA$ pour $A \in \mathcal{C}_0$ telle que pour tout morphisme $f \in \mathcal{C}[A, B]$, le diagramme suivant commute :

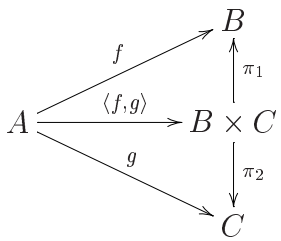
$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle \eta_A}\downarrow & & \downarrow{\scriptstyle \eta_B} \\ GA & \xrightarrow{Gf} & FB \end{array}$$

Un isomorphisme naturel est une transformation naturelle η telle que pour chaque A, η_A est un isomorphisme.

Définition 1.2.6 (Catégorie cartésienne)

Une catégorie $\mathcal{C}$ est dite cartésienne s'il existe :
- un objet terminal 1, c'est à dire tel que pour tout object $A \in \mathcal{C}_0$, il existe un unique morphisme $!_A$ de $A \to 1$,
- un bifoncteur $- \times - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$,
- pour chaque paire d'objets A, B des projections $\pi_i^{A,B} : A_1 \times A_2 \to A_i$ $(i = 1, 2)$

tels que pour toute paire de morphismes de même source $f : A \to B$, $g : A \to C$, il existe un unique morphisme $\langle f, g\rangle : A \to B \times C$ tel que



Définition 1.2.7 (Adjonction)

Une adjonction entre deux foncteurs $L : \mathcal{C} \to \mathcal{D}$ et $R : \mathcal{D} \to \mathcal{C}$ est un isomorphisme naturel $\tau : \mathcal{D}[L-,-] \to \mathcal{C}[-,R-]$. On dit que L est adjoint à gauche de R.

Définition 1.2.8 (Catégorie cartésienne fermée)

Une catégorie cartésienne fermée est une catégorie cartésienne $\langle \mathcal{C}, \times\rangle$ munie d'un bifoncteur $- \Rightarrow - : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ tel que pour tous objets A, B, $- \times A$ est adjoint à gauche de $A \Rightarrow -$. On note $\Lambda^{A,B,C} : \mathcal{C}[A \times B, C] \to \mathcal{C}[A, B \Rightarrow C]$ (curryfication) et $V : \mathcal{C}[A, B \Rightarrow C] \to \mathcal{C}[A \times B, C]$ (décurryfication) les isomorphismes naturels.

On notera $\mathsf{ev}^{A,B} = V^{A\Rightarrow B,A,B}(\mathsf{id}_{A\Rightarrow B})$, un morphisme de $(A \Rightarrow B) \times A \to B$.

Lemme 1.2.9

Pour tous $f : C \times A \to B$ et $g : C \to A$, $\mathsf{ev}^{A,B} \circ \langle \Lambda f, g\rangle = f \circ \langle \mathsf{id}_C, g\rangle$

Démonstration :

$$\begin{aligned} \mathsf{ev}^{A,B} \circ \langle \Lambda f, g\rangle &= V(\mathsf{id}_{A\Rightarrow B}) \circ (\Lambda f \times \mathsf{id}_A) \circ \langle \mathsf{id}_C, g\rangle \\ &= V(\Lambda f) \circ \langle \mathsf{id}_C, g\rangle \\ &= f \circ \langle \mathsf{id}_C, g\rangle \end{aligned}$$ ■

1.2.2 Catégories CPO-enrichies

Les catégories cartésiennes fermées forment des modèles pour le λ-calcul simplement typé sans récursion (avec une abstraction simple). Pour représenter les fonctions récursives, il est nécessaire d'avoir une structure plus forte sur les domaines : avec des ordres partiels complets, on va pouvoir définir une suite croissante, et représenter la dénotation d'une fonction récursive comme sa limite.

Définition 1.2.10 (Ensemble dirigé)

Un ensemble dirigé d'un ordre partiel $\langle X, \sqsubseteq \rangle$ est une partie non vide $D \subseteq X$ telle que

$$\forall x, y \in D, \exists z \in D, x \sqsubseteq z \wedge y \sqsubseteq z$$

Définition 1.2.11 (Ordre partiel complet)

Un ordre partiel complet est un triplet tel que $\langle X, \sqsubseteq, \bot \rangle$ tel que
- $\langle X, \sqsubseteq \rangle$ est un ordre partiel,
- $\bot$ est le plus petit élément de $\langle X, \sqsubseteq \rangle$,
- tous les ensembles dirigés $D \subseteq X$ admettent une borne supérieure $\bigsqcup D$.

Définition 1.2.12 (Ordre partiel complet plat, soulèvement)

Si A est un ensemble, on définit $A_\bot$ comme l'ensemble $A \cup \{\bot\}$ (union disjointe) équipé de l'ordre

$$x \sqsubseteq y \iff (x = y) \ \vee \ (x = \bot)$$

On notera $N_\bot$ (resp. $B_\bot$, $1_\bot$) le soulèvement de l'ensemble $\{0, 1, 2 \ldots\}$ (resp. $\{\mathrm{tt}, \mathrm{ff}\}$, $\{\top\}$).

Définition 1.2.13 (Fonction continue)

Soient $\langle X, \sqsubseteq_X \rangle$ et $\langle Y, \sqsubseteq_Y \rangle$ deux ordres partiels complets. Une application $f : X \to Y$ est continue si pour tout ensemble dirigé $D \subseteq X$, $f(D)$ est dirigé et $f(\bigsqcup D) = \bigsqcup f(D)$.

Définition 1.2.14 (Catégorie CPO-enrichie)

Une catégorie $\mathcal{C}$ est dite CPO enrichie si pour tous A, B, $\mathcal{C}[A, B]$ est un ordre partiel complet et si la composition est continue.

Une catégorie cartésienne fermée est dite CPO-enrichie si les opérations de paire $\langle -, - \rangle$, de curryfication et de décurryfication sont continues.

1.2.3 Modèles et modèles standard

Définition 1.2.15 (Modèle du $\boldsymbol{\lambda}$-calcul simplement typé)

Un modèle du λ-calcul simplement typé est la donnée de :
- une catégorie cartésienne fermée CPO-enrichie $\mathcal{C}$,

– un objet $[\![\iota]\!]$ de $\mathcal{C}$,

On définit $[\![A \to B]\!] = [\![A]\!] \Rightarrow [\![B]\!]$ et, si Γ est un environnement de typage $x_1 : A_1 \dots x_n : A_n$,

$$[\![\Gamma]\!] = (\dots(1 \times [\![A_1]\!]) \times \dots) \times [\![A_n]\!]$$

Pour toute preuve de $\Gamma \vdash M : B$, on définit une dénotation

$$[\![\Gamma]\!] \xrightarrow{[\![x_1:A_1 \dots x_n:A_n \vdash M:B]\!]} [\![B]\!]$$

par

– $[\![x_1 : A_A \dots x_n : A_n \vdash x_i : A_i]\!] = \pi_2 \circ \pi_1^n$ (on notera plus simplement π_i quand il n'y a pas de confusion possible avec les morphismes π_1 et π_2 de la structure cartésienne)

– $[\![\Gamma \vdash \mathsf{rec}\ f\ x = M : A \to B]\!] = \bigsqcup \varphi_i$ où φ_n est une famille croissante de $\mathcal{C}[[\![\Gamma]\!], [\![A \to B]\!]]$ définie par

$$\begin{cases} \varphi_0 = \bot \\ \varphi_{n+1} = \Lambda([\![\Gamma, x : A, f : A \to B \vdash M : B]\!] \circ \langle \mathsf{id}_{\Gamma \times A}, \varphi_n \circ \pi_1 \rangle) \end{cases}$$

– $[\![\Gamma \vdash (M)N : B]\!] = \mathsf{ev} \circ \langle [\![\Gamma \vdash M : A \to B]\!], [\![\Gamma \vdash N : A]\!] \rangle$

Il faut noter que, dans toute la suite de cette thèse, on parlera de la dénotation d'un séquent $\Gamma \vdash P : A$, alors qu'il s'agit véritablement de la dénotation de sa preuve : comme la dérivation de typage est unique, il n'y a en fait aucune ambiguïté.

Définition 1.2.16 (Modèle d'un **λ**-calcul étendu)

Un modèle d'un λ calcul étendu est un modèle du λ-calcul simplement typé avec pour chaque constante c un morphisme $\underline{c} : 1 \to [\![A(c)]\!]$. On étend la définition des dénotations par $[\![\Gamma \vdash c : A(c)]\!] = \underline{c} \circ !$. Les δ-règles doivent être validées :

$$\forall c, c', \begin{cases} A(c) = \iota \to A \\ A(c') = \iota \end{cases} \implies \mathsf{ev} \circ \langle \underline{c}, \underline{c'} \rangle = [\![\vdash \mathsf{eval}_\delta(c, c') : A]\!]$$

On commence par prouver quelques lemmes utiles sur les dénotations.

Lemme 1.2.17 (Extension du contexte)

Pour tout terme P où x n'est pas libre, $[\![\Gamma, x : A \vdash P : B]\!] = [\![\Gamma \vdash P : B]\!] \circ \pi_1$

Démonstration : Par induction sur la dérivation de $\Gamma \vdash P : B$.

■

Lemme 1.2.18 (La composition correspond à la substitution)

Pour tout terme P, $[\![\Gamma, x : A \vdash P : B]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle = [\![\Gamma \vdash P[x \backslash Q] : B]\!]$

Démonstration : Par induction sur la dérivation de $\Gamma, x : A \vdash P : B$. Si $P = c$, alors $P[x\backslash Q] = c$ et

$$\begin{array}{rcl} [\![\Gamma, x : A \vdash c : B]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle & = & \underline{c} \circ ! \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ & = & \underline{c} \circ ! \\ & = & [\![\Gamma \vdash c : B]\!] \end{array}$$

Si $P = x$, alors $P[x\backslash Q] = Q$ et

$$\begin{array}{rcl} [\![\Gamma, x : A \vdash x : A]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle & = & \pi_2 \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ & = & [\![\Gamma \vdash Q : A]\!] \end{array}$$

Si $P = y \neq x$, alors $P[x\backslash Q] = y$ et

$$\begin{array}{rcl} [\![\Gamma, x : A \vdash y : B]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle & = & \pi_2 \circ \pi_1^{n+1} \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ & = & \pi_2 \circ \pi_1^n \\ & = & [\![\Gamma \vdash y : B]\!] \end{array}$$

Si $P = (P')P''$, alors $P[x\backslash Q] = (P'[x\backslash Q])P''[x\backslash Q]$ et

$$\begin{array}{l} [\![\Gamma, x : A \vdash P : B]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ \quad = \mathsf{ev} \circ \langle [\![\Gamma, x : A \vdash P' : C \to B]\!], [\![\Gamma, x : A \vdash P'' : C]\!] \rangle \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ \quad = \mathsf{ev} \circ \langle [\![\Gamma, x : A \vdash P' : C \to B]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ \qquad\quad [\![\Gamma, x : A \vdash P'' : C]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \rangle \\ \quad = \mathsf{ev} \circ \langle [\![\Gamma \vdash P'[x\backslash Q] : C \to B]\!], [\![\Gamma \vdash P''[x\backslash Q] : C]\!] \rangle \\ \quad = [\![\Gamma \vdash (P'[x\backslash Q])P''[x\backslash Q] : B]\!] \\ \quad = [\![\Gamma \vdash P[x\backslash Q] : B]\!] \end{array}$$

Si $P = \mathsf{rec}\ f\ y = P'$ alors on définit

$$\begin{array}{l} \varphi_0 = \bot \\ \varphi_{n+1} = \Lambda\,([\![\Gamma, x, y, f \vdash P']\!] \circ \langle \mathsf{id}_{\Gamma,x,y}, \varphi_n \circ \pi_1, \rangle) \\ \phi_0 = \bot \\ \phi_{n+1} = \Lambda\,([\![\Gamma, y, f \vdash P'[x\backslash Q]]\!] \circ \langle \mathsf{id}_{\Gamma,y}, \phi_n \circ \pi_1, \rangle) \end{array}$$

On a $[\![P]\!] = \bigsqcup \varphi_n$ et $[\![P[x\backslash Q]]\!] = \bigsqcup \phi_n$. Il suffit de prouver que pour tout n $\phi_n = \varphi_n \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle$. C'est évident pour $n = 0$.

$$\begin{array}{l} \varphi_{n+1} \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ \quad = \Lambda\,([\![\Gamma, x, y, f \vdash P']\!] \circ \langle \mathsf{id}_{\Gamma,x,y}, \varphi_n \circ \pi_1 \rangle) \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\ \quad = \Lambda\,([\![\Gamma, x, y, f \vdash P']\!] \circ \langle \mathsf{id}_{\Gamma,x,y}, \varphi_n \circ \pi_1 \rangle \circ (\langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \times \mathsf{id}_y)) \end{array}$$

On a

$$\begin{aligned}
&\langle \mathsf{id}_{\Gamma,x,y}, \varphi_n \circ \pi_1 \rangle \circ (\langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \times \mathsf{id}_y) \\
&\quad = \langle \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \times \mathsf{id}_y, \varphi_n \circ [\![\Gamma \vdash Q : A]\!] \circ \pi_1 \rangle \\
&\quad = \langle \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \times \mathsf{id}_y, \phi_n \circ \pi_1 \rangle \\
&\quad = (\langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \times \mathsf{id}_y \times \mathsf{id}_x) \circ \langle \mathsf{id}_{\Gamma,y}, \phi_n \circ \pi_1 \rangle \\
&\quad = \alpha \circ (\langle \mathsf{id}_{\Gamma,y,f}, [\![\Gamma \vdash Q : A]\!] \circ \pi_1 \circ \pi_1 \rangle) \circ \langle \mathsf{id}_{\Gamma,y}, \phi_n \circ \pi_1 \rangle
\end{aligned}$$

où α est l'isomorphisme de la catégorie cartésienne qui transforme $[\![\Gamma, x, y, f]\!]$ en $[\![\Gamma, y, f, x]\!]$:

$$[\![\Gamma]\!] \times [\![A]\!] \times [\![C]\!] \times [\![C \to B]\!] \xrightarrow{\alpha} [\![\Gamma]\!] \times [\![C]\!] \times [\![C \to B]\!] \times [\![A]\!]$$

D'une part, $[\![\Gamma \vdash Q : A]\!] \circ \pi_1 \circ \pi_1 = [\![\Gamma, y, f \vdash Q : A]\!]$ et d'autre part

$$[\![\Gamma, x, y, f \vdash P' : B]\!] \circ \alpha = [\![\Gamma, y, f, x \vdash P' : B]\!]$$

On obtient

$$\begin{aligned}
&\varphi_{n+1} \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q : A]\!] \rangle \\
&\quad = \Lambda\,([\![\Gamma, y, f, x \vdash P' : B]\!] \circ \langle \mathsf{id}_{\Gamma,y,f}, [\![\Gamma, y, f \vdash Q : A]\!] \rangle \circ \langle \mathsf{id}_{\Gamma,y}, \phi_n \circ \pi_1 \rangle) \\
&\quad = \Lambda\,[\![\Gamma, y, f \vdash P'[x\backslash Q] : B]\!] \circ \langle \mathsf{id}_{\Gamma,y}, \phi_n \circ \pi_1 \rangle) \\
&\quad = \phi_{n+1}
\end{aligned}$$

ce qui conclut la preuve. ■

Ce lemme permet de prouver que la sémantique modélise correctement la β-réduction (dans sa forme récursive).

Lemme 1.2.19 (β-réduction)

Pour tous $\Gamma, f : A \to B, x : A \vdash P : B$ et $\Gamma \vdash Q : A$, on a

$$\mathsf{ev} \circ \langle [\![\Gamma \vdash \mathsf{rec}\ f\ x = P : A \to B]\!], [\![\Gamma \vdash Q : A]\!] \rangle = [\![\Gamma \vdash P[f\backslash \mathsf{rec}\ f\ x = P, x\backslash Q]]\!]$$

Démonstration : On calcule

$$\begin{aligned}
&\mathsf{ev} \circ \langle [\![\Gamma \vdash \mathsf{rec}\ f\ x = P]\!], [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = \mathsf{ev} \circ \langle \bigsqcup \varphi_n, [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = \mathsf{ev} \circ \langle \bigsqcup \varphi_{n+1}, [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = \mathsf{ev} \circ \langle \bigsqcup \Lambda([\![\Gamma, x, f \vdash P]\!] \circ \langle \mathsf{id}_\Gamma, \varphi_n \circ \pi_1 \rangle), [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = \bigsqcup \mathsf{ev} \circ \langle \Lambda([\![\Gamma, x, f \vdash P]\!] \circ \langle \mathsf{id}_\Gamma, \varphi_n \circ \pi_1 \rangle), [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = \bigsqcup ([\![\Gamma, x, f \vdash P]\!] \circ \langle \mathsf{id}_{\Gamma,x}, \varphi_n \circ \pi_1 \rangle) \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = [\![\Gamma, x, f \vdash P]\!] \circ \langle \mathsf{id}_{\Gamma,x}, \bigsqcup \varphi_n \circ \pi_1 \rangle \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = [\![\Gamma, x, f \vdash P]\!] \circ \langle \mathsf{id}_{\Gamma,x}, [\![\Gamma \vdash \mathsf{rec}\ f\ x = P]\!] \circ \pi_1 \rangle \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q]\!] \rangle \\
&\quad = [\![\Gamma, x, f \vdash P]\!] \circ \langle \mathsf{id}_{\Gamma,x}, [\![\Gamma, x \vdash \mathsf{rec}\ f\ x = P]\!] \rangle \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash Q]\!] \rangle
\end{aligned}$$

En appliquant deux fois le lemme précédent, on obtient

$$\mathsf{ev} \circ \langle [\![\mathsf{rec}\ f\ x = P]\!], [\![Q]\!] \rangle = [\![\Gamma \vdash P[f\backslash \mathsf{rec}\ f\ x = P, x\backslash Q]]\!]$$

■

1.2.4 Correction et adéquation

Nous avons maintenant les outils nécessaires pour prouver que la sémantique dénotationnelle calcule effectivement le résultat obtenu par la sémantique à grand pas.

Théorème 1.2.20 (Correction)

Si $\mathcal{M}$ est un modèle, alors

$$P \Downarrow Q \Longrightarrow [\![\Gamma \vdash P : A]\!] = [\![\Gamma \vdash Q : A]\!]$$

Démonstration : Par induction sur la preuve de $P \Downarrow Q$. Si $P \Downarrow Q$ est un axiome, alors $P = Q$ et la conclusion est triviale. Pour la règle

$$\frac{P \Downarrow \mathsf{rec}\ f\ x = P' \qquad P'[x\backslash Q, f\backslash \mathsf{rec}\ f\ x = P'] \Downarrow V}{(P)Q \Downarrow V}$$

on a, par hypothèse d'induction $[\![P]\!] = [\![\mathsf{rec}\ f\ x = P']\!]$ et $[\![P'[x\backslash Q, f\backslash \mathsf{rec}\ f\ x = P']]\!] = [\![V]\!]$. On calcule, en appliquant le lemme 1.2.19,

$$\begin{array}{rcl} [\![(P)Q]\!] & = & \mathsf{ev} \circ \langle [\![P]\!], [\![Q]\!] \rangle \\ & = & \mathsf{ev} \circ \langle [\![\mathsf{rec}\ f\ x = P']\!], [\![Q]\!] \rangle \\ & = & [\![\Gamma \vdash P'[f\backslash \mathsf{rec}\ f\ x = P', x\backslash Q] : B]\!] \\ & = & [\![\Gamma \vdash V : B]\!] \end{array}$$

Pour la règle

$$\frac{P \Downarrow c \qquad Q \Downarrow c'}{(P)Q \Downarrow \mathsf{eval}_\delta(c, c')}$$

on a par hypothèse

$$[\![\Gamma \vdash P : A \to B]\!] = [\![\Gamma \vdash c : A \to B]\!] = \underline{c} \circ !$$
$$[\![\Gamma \vdash Q : A]\!] = [\![\Gamma \vdash c' : A]\!] = \underline{c'} \circ !$$

On calcule

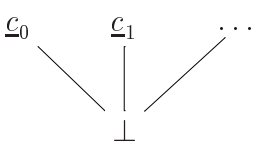
$$\begin{array}{rcl} [\![(P)Q]\!] & = & \mathsf{ev} \circ \langle [\![P]\!], [\![Q]\!] \rangle \\ & = & \mathsf{ev} \circ \langle \underline{c}, \underline{c'} \rangle \circ ! \\ & = & [\![\vdash \mathsf{eval}_\delta(c, c') : B]\!] \circ ! \\ & = & [\![\Gamma \vdash \mathsf{eval}_\delta(c, c') : B]\!] \end{array}$$

■

La réciproque n'est pas vraie en général. Il faut interpréter le type de base ι et les constantes qui agissent dessus de façon raisonnable.

Définition 1.2.21 (Modèle standard)

Un modèle standard est un modèle tel que $\mathcal{C}[1, [\![\iota]\!]]$ est l'ordre partiel plat



où $c_0, c_1, \ldots$ sont les constantes de type ι, et tel que pour toute constante $c : \iota \to A$, $\mathsf{ev} \circ \langle \underline{c}, f \rangle \neq \bot$ implique que $f = \underline{c'}$ avec c' tel que $\mathsf{eval}_\delta(c, c')$ est défini. Autrement dit, les dénotations des constantes (hors type de base) ne sont pas plus définies que la syntaxe ne l'exige.

Ces modèles valident la réciproque du théorème de correction, dans le sens suivant :

Théorème 1.2.22 (Adéquation)

Dans un modèle standard, si $[\![\vdash P : A]\!] \neq \bot$, alors il existe Q tel que $P \Downarrow Q$.

Démonstration : Par induction sur P. Dans le cas des constantes et des abstractions, $P \Downarrow P$, donc la propriété est triviale. Le cas des variables est exclu car P est fermé.

Pour le cas de l'application, on écrit $P = (P')P''$. $[\![\vdash P : A]\!] = \mathsf{ev} \circ \langle [\![\vdash P' : B \to A]\!], [\![\vdash P'' : B]\!] \rangle$. Il apparaît que $[\![\vdash P' : B \to A]\!] \neq \bot$, et donc, par hypothèse d'induction, qu'il existe Q' tel que $P' \Downarrow Q'$ avec, par le thèorème de correction, $[\![\vdash P' : B \to A]\!] = [\![\vdash Q' : B \to A]\!]$.

Si $Q' = \mathsf{rec}\ f\ x = R'$, alors, par le lemme 1.2.19,

$$\begin{array}{rcl} [\![\vdash P : A]\!] & = & \mathsf{ev} \circ \langle [\![\vdash \mathsf{rec}\ f\ x = R' : B \to A]\!], [\![\vdash P'' : B]\!] \rangle \\ & = & [\![\vdash R'[f \backslash \mathsf{rec}\ f\ x = R', x \backslash P''] : A]\!] \end{array}$$

Comme $[\![\vdash P : A]\!] \neq \bot$, il existe Q'' tel que $R'[f \backslash \mathsf{rec}\ f\ x = R', x \backslash P''] \Downarrow Q''$. On a $[\![\vdash Q'' : A]\!] = [\![\vdash P : A]\!]$. Pour conclure, il suffit de rappeler la règle pour l'élimination de l'abstraction :

$$\frac{P' \Downarrow \mathsf{rec}\ f\ x = R' \qquad R'[f \backslash \mathsf{rec}\ f\ x = R', x \backslash P''] \Downarrow Q''}{(P')P'' \Downarrow Q''}$$

Si P' est une constante c', alors $B = \iota$ et, comme le modèle est standard, $[\![\vdash P'' : B]\!] = \underline{c''}$, avec $\mathsf{eval}_\delta(c', c'')$ défini. Ainsi, il existe Q'' tel que $P'' \Downarrow Q''$. Par correction, $[\![\vdash Q'' : \iota]\!] = \underline{c''}$. Comme les seules valeurs au type de base sont des constantes, et que le modèle est standard, $Q'' = c''$. On obtient donc $[\![\vdash P : A]\!] = [\![\mathsf{eval}_\delta(c', c'')]\!]$ et

$$\frac{P' \Downarrow c' \qquad P'' \Downarrow c''}{(P')P'' \Downarrow \mathsf{eval}_\delta(c', c'')}$$

■

1.3 Modèles usuels

Dans ce chapitre, on rappelle les définitions de quelques modèles standard qui serviront dans la suite. Pour plus de précisions, on pourra se reporter à [AC98].

1.3.1 Domaines de Scott

Ordres partiels complets

Proposition 1.3.1

La catégorie des ordres partiels complets et les applications continues, qu'on notera Cpo, forme un modèle standard de PCF (resp. PCF finitaire, PCF unaire).

Démonstration : Évidemment, Cpo est CPO-enrichie. Il est aisé de vérifier que la composée de deux fonctions continues est continue : Cpo est donc une sous-catégorie de Set. Comme la paire de deux fonctions continues est aussi continue, Cpo est une catégorie cartésienne.

On définit l'exponentielle $A \to B$ comme l'ensemble des applications continues de A dans B, ordonnées pour l'ordre point à point. On prouve que $A \to B$ est aussi un CPO.

Si $f : A \times B \to C$ est continue, on définit $(Vf)(b) : a \mapsto f(a, b)$, et réciproquement si $f : B \to A \to C$ est continue, on définit $(\Lambda f)(a, b) = (f(a))(b)$.

Il est facile de vérifier que Λ et V sont inverses, et que leurs images sont des fonctions continues et qu'en tant qu'applications entre domaines de fonctions, elles ont également continues.

On choisit bien sûr pour l'objet de base $\mathsf{N}_\perp$ pour PCF, $\mathsf{B}_\perp$ pour PCF finitaire et $\mathsf{U}_\perp$ pour PCF unaire. ■

Compacité et algébricité

Les fonctions continues sont un mauvais modèle de PCF : en effet, elles ne représentent en rien l'intuition qui veut qu'une fonction utilise une partie finie de son argument pour produire un résultat fini. Le concept même de finitude n'est pas clair. On définit donc les compacts, qui sont les éléments finis, et l'algébricité, qui exprime qu'un élément (infini) est la somme de ses parties finies.

Définition 1.3.2 (Compacité, Algébricité)

Soit $\langle X, \sqsubseteq \rangle$ un ordre partiel complet. Un élément $x \in X$ est compact si, pour tout ensemble dirigé $A \subseteq X$

$$x \sqsubseteq \bigsqcup A \implies \exists a \in A, x \sqsubseteq a$$

On note $\mathcal{K}(X)$ l'ensemble des éléments compacts de X. X est dit algébrique, si pour tout $x \in X$, l'ensemble $\{y \in \mathcal{K}(X) \mid y \sqsubseteq x\}$ est dirigé et

a pour borne supérieure x.

Définition 1.3.3 (Domaine de Scott)

Un domaine de Scott est un ordre partiel complet algébrique $\langle X, \sqsubseteq \rangle$ tel que pour tout $x, y \in X$, si x et y sont bornés (on écrit $x \uparrow y$), alors $x \sqcup y$ (la borne supérieure de x et y) existe.

Proposition 1.3.4

Les domaines de Scott et les fonctions continues forment des modèles de PCF, PCF finitaire et PCF unaire.

1.3.2 Stabilité et cohérence

Les domaines de Scott, s'ils représentent fidèlement la notion de finitude de l'information, modélisent assez mal l'aspect séquentiel des langages considérés. Ainsi, beaucoup de fonctions de $(B_\perp \times B_\perp) \to B_\perp$ ne sont pas définissables, car elles dépendent de leurs deux arguments de la même façon, alors qu'un programme devra tester l'un avant l'autre. Pour modéliser cet index de séquentialité, Berry [Ber79] a imaginé le modèle des fonctions stables.

Les ordres stables seront notés $\leq$, par opposition aux ordres extensionnels qu'on notera $\sqsubseteq$.

Définition 1.3.5 (dI-domaine)

Un dI-domaine est un domaine de Scott $\langle X, \leq \rangle$ tel que :
- pour tout $x \in \mathcal{K}(X)$, $\{y \in X \mid y \leq x\}$ est fini.
- si $x \uparrow y$, alors $x \wedge y$ existe,
- si x, y, z ont un majorant commun dans X, $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$,

Définition 1.3.6 (Fonction stable)

Une fonction stable entre deux dI-domaines X and Y est une fonction continue f telle que

$$\forall x, y \in X, x \uparrow y \Longrightarrow f(x \wedge y) = f(x) \wedge f(y)$$

Définition 1.3.7 (Ordre stable)

Soient $f, g : X \to Y$ deux fonctions stables. $f \leq g$ si

$$\forall x \leq y, f(x) = f(y) \wedge g(x)$$

Proposition 1.3.8

La catégorie des dI-domaines et des fonctions stables est cartésienne fermée, et forme un modèle de PCF (resp. de PCF finitaire, de PCF unaire).

Une façon pratique de représenter les fonctions stables est d'utiliser leur trace : chaque petit morceau de la réponse (un élément premier) utilise une partie finie des entrées (un élément compact).

Proposition 1.3.9

Une fonction continue f entre deux dI-domaines X et Y est stable si et seulement si pour tous $x \in X$, et $b \in \mathcal{K}(Y)$, tels que $b \leq f(x)$, il existe $a \in \mathcal{K}(X)$ minimal tel que $a \leq x$ et $b \leq f(a)$.

Définition 1.3.10 (Élément premier)

Un élément x d'un CPO $\langle X, \sqsubseteq \rangle$ est premier si pour tout $Y \subseteq X$ non vide fini tel que $\bigsqcup Y$ existe,

$$x \sqsubseteq \bigsqcup Y \implies \exists y \in Y, x \sqsubseteq y$$

On note $|X|$ l'ensemble des éléments premiers de X.

Définition 1.3.11 (Trace)

Si f est une fonction stable, sa trace $\mathsf{tr}(f)$ est l'ensemble des couples (a, b) où
- $a \in \mathcal{K}(X)$ et $b \in |Y|$
- a est minimal tel que $b \leq f(a)$

Proposition 1.3.12

Si $f, g : X \to Y$ sont deux fonctions stables, $\mathsf{tr}(f) = \mathsf{tr}(g)$ implique $f = g$, et $\mathsf{tr}(f) \subseteq \mathsf{tr}(g)$ implique $f \leq g$.

Bidomaines

Malheureusement, les fonctions stables ne sont pas toutes monotones, et donc ne sont pas toutes définissables. Pour tenter d'y remédier, Berry a proposé un modèle qui combine les deux contraintes : un ordre stable et un ordre extensionnel.

Définition 1.3.13

$\langle D, \sqsubseteq, \leq, \bot \rangle$ est un bidomaine si :
- $\langle D, \sqsubseteq, \bot \rangle$ est un domaine de Scott,

- $\langle D, \leq, \bot \rangle$ est un dI-domaine,
- l'identité de $\langle D, \sqsubseteq, \bot \rangle$ vers $\langle D, \leq, \bot \rangle$ est continue,
- si x et y sont bornés dans $\langle D, \leq, \bot \rangle$ alors $x \wedge y = x \sqcap y$.

On rappelle également la définition du domaine flèche : $D \Rightarrow D'$:
- $f \in D \Rightarrow D'$ si f est stable pour $\leq$ et continue pour $\leq$ et $\sqsubseteq$,
- $f \sqsubseteq g$ si $\forall x \in D, fx \sqsubseteq gx$
- $f \leq g$ si $\forall x, y \in D, x \leq y \Longrightarrow f(x) = f(y) \wedge g(x)$

Proposition 1.3.14

La catégorie des bidomaines est cartésienne fermée.

1.3.3 Stabilité forte

Il reste que certains éléments ne sont pas définissables, même au premier ordre : ainsi la fonction $f : B_\bot^3 \to B$ définie par sa trace

$$\begin{cases} f\ \mathrm{tt}\,\mathrm{ff}\,\bot = \mathrm{tt} \\ f\ \bot\,\mathrm{tt}\,\mathrm{ff} = \mathrm{tt} \\ f\ \mathrm{ff}\,\bot\,\mathrm{tt} = \mathrm{tt} \end{cases}$$

est stable, monotone, mais pas définissable.

La stabilité forte est une généralisation de la stabilité : au lieu de considérer les paires d'éléments cohérents (bornés), on considère les ensembles finis d'éléments cohérents. Nous rappelons ici seulement les définitions. Pour plus de détails, on pourra se reporter aux travaux de Bucciarelli et Ehrhard [BE93, BE94].

Définition 1.3.15 (Ordre d'Egli-Milner)

Soit $\langle X, \leq \rangle$ un ordre partiel, $A, B \subseteq X$. On dit que A est Egli-Milner plus petit que B, et on note $A \sqsubseteq B$ si
- $\forall a \in A, \exists b \in B, a \leq b$, et
- $\forall b \in B, \exists a \in A, a \leq b$.

Définition 1.3.16 (dI-domaine avec cohérence)

Un dI-domaine avec cohérence (dIC) est un dI-domaine $\langle X, \leq \rangle$ muni d'une famille de sous ensembles finis $C(X)$ telle que :
- pour tout $x \in X$, $\{x\} \in C(X)$,
- si A est un sous ensemble fini de X et $B \in C(X)$, alors $A \sqsubseteq B \Longrightarrow A \in C(X)$,
- si $D_1 \ldots D_n$ sont des ensembles dirigés de X, et si pour tout $x_1 \in D_1 \ldots x_n \in D_n$, $\{x_1, \ldots, x_n\} \in C(X)$, alors $\{\bigsqcup D_1, \ldots, \bigsqcup D_n\} \in$

$C(X)$ aussi.

Définition 1.3.17 (Fonction fortement stable)

Une fonction fortement stable entre deux dIC X et Y est une fonction continue f telle que, pour tout $A \in C(X)$, $f(A) \in C(Y)$ et $f(\bigwedge A) = \bigwedge f(A)$.

Définition 1.3.18 (Appariement)

Un appariement de X et Y est une relation $R \subseteq X \times Y$ telle que $\pi_1(R) = X$ et $\pi_2(R) = Y$ (R et R^{-1} sont surjectives).

Définition 1.3.19 (Cohérence linéaire)

- $A \in C(N_\perp)$ est (linéairement) cohérent s'il contient $\perp$ ou si c'est un singleton.
- Le dI-domaine avec cohérence produit $X \times Y$ est le produit des domaines, et $A \in C(X \times Y)$ est cohérent si $\pi_1(A) \in C(X)$ et $\pi_2(A) \in C_Y$.
- le dI-domaine flèche $X \Rightarrow Y$ est l'ensemble des fonctions fortement stables de X dans Y, équipé de la cohérence : $F \in C(X \Rightarrow Y)$ si pour tout $A \in C(X)$ et tout appariement $\mathcal{E}$ de F et de A $\bigsqcup\{f(a) \mid (f,a) \in \mathcal{E}\} = (\bigsqcup F)(\bigsqcup A)$

On verra plus loin que la notion de cohérence linéaire est liée aux relations de Sieber.

Théorème 1.3.20

La catégorie des dI-domaines avec cohérence et des fonctions fortement stables forment un modèle de PCF.

Théorème 1.3.21 (Définissabilité au premier ordre)

Pour PCF finitaire, toutes les fonctions fortement stables du premier ordre sont séquentielles, donc définissables.

Ce n'est plus vrai aux ordres supérieurs, comme on le verra dans le chapitre suivant, ni pour PCF, puisqu'il n'y a dans ces modèles aucune notion de calculabilité, et qu'il est bien connu que toutes les fonctions des entiers dans les entiers ne sont pas calculables. Ce théorème indique en fait que la stabilité forte capture la notion de séquentialité au premier ordre.

1.3.4 Jeux de Hyland et Ong

La sémantique des jeux combine des idées venant des algorithmes séquentiels de Berry et Curien [BC82] et de la sémantique de traces des calculs de

processus. Elle a permis de construire les premiers modèles de PCF complètement adéquats non syntaxiques [AJM00] et [HO00]. Elle a depuis été reformulée pour offrir un cadre assez général pour l'étude d'une large gamme d'extensions de PCF : avec des opérateurs de contrôles [Lai98], avec des références simple (Idealized Algol) ou générales [AHM98], du non-déterminisme [Har99] et de la concurrence [GM04].

On reprendra ici les définitions de Harmer [Har99], dérivées de celles de Hyland et Ong [HO00].

Définition 1.3.22 (Arène)

Une arène est un triplet $\langle M_A, \lambda_A, \vdash_A \rangle$ où
- M_A est un ensemble de coups,
- λ_A est la fonction d'étiquetage, de M_A dans $\{O, P\} \times \{Q, A\}$, qui associe à chaque coup une polarité (opposant ou joueur) et précise si c'est une question ou une réponse,
- $\vdash_A$ est la relation d'activation, telle que
  - si $n \vdash_A m$ et $n \neq m$, alors $\lambda_A^{O,P}(n) \neq \lambda_A^{O,P}(m)$ ($\lambda_A^{O,P}(n)$ est la projection de $\lambda_A(n)$ sur l'axe O, P)
  - si $m \vdash_A m$, alors $\lambda_A(m) = OQ$ et il n'existe pas de $n \neq m$ tel que $n \vdash_A m$ (m est un coup initial),
  - si $n \vdash_A m$ et si m est une réponse, alors n est une question.

Définition 1.3.23 (Partie)

Une suite pointée dans une arène A est une suite finie de M_A, avec un pointeur de chaque coup m vers un coup précédent n tel que $n \vdash_A m$. Le premier ancêtre d'un coup m est l'unique coup initial m_0 tel qu'il existe une sous-suite $m_0 m_1 \dots m_k$ où chaque m_{i+1} pointe sur m_i et $m = m_k$.

Une partie est une suite pointée où l'opposant commence et où le joueur et l'opposant alternent. On note L_A l'ensemble des parties.

Définition 1.3.24 (Fil, vue)

Si sm est de longueur impaire, le fil $\lceil sm \rceil$ est la sous-suite des ancêtres de m (la composante connexe de m). Si sm est de longueur paire, on dit que sm respecte le fil en m si m pointe dans $\lceil s \rceil$. Une partie respecte les fils si elle respecte le fil à chaque coup du joueur. Les fils d'une parties sont aussi des parties.

La vue du joueur, notée $\ulcorner - \urcorner$ (à ne pas confondre avec le fil $\lceil - \rceil$) est définie par :
- $\ulcorner sm \urcorner = m$ si m est un coup initial,
- $\ulcorner smtn \urcorner = \ulcorner s \urcorner mn$ si n est un coup de l'opposant qui pointe sur m,

– $\ulcorner sm \urcorner = \ulcorner s \urcorner m$ si m est une coup du joueur.

Une partie $sm \in L_A^{paire}$ respecte la visibilité en m si m pointe dans $\ulcorner s \urcorner$. Elle respecte la visibilité si elle respecte la visibilité à chaque coup joueur.

Définition 1.3.25 (Constructions sur les arènes)

L'arène produit $A \times B$:
- $M_{A\times B} = M_A + M_B$,
- $\lambda_{A\times B}(m_A) = \lambda_A(m_A)$ et $\lambda_{A\times B}(m_B) = \lambda_B(m_B)$,
- $m \vdash_{A\times B} n$ si m, n sont des coups dans A et $m \vdash_A n$ ou si m, n sont des coups dans B et $m \vdash_B n$.

L'arène flèche $A \Rightarrow B$:
- $M_{A\Rightarrow B} = M_A + M_B$,
- $\lambda_{A\Rightarrow B}(m_A) = \bar{\lambda}_A(m_A)$ et $\lambda_{A\Rightarrow B}(m_B) = \lambda_B(m_B)$,
- $m \vdash_{A\times B} n$ si m, n sont des coups dans A et $m \vdash_A n$ ou si m, n sont des coups dans B et $m \vdash_B n$, ou si m est un coup de B et n est un coup initial de A.

L'arène soulevée $A_\perp$
- $M_{A_\perp} = \{q, a\} + M_A$,
- $\lambda_{A_\perp}(q) = OQ$, $\lambda_{A_\perp}(a) = PA$, $\lambda_{A_\perp}(m) = \lambda_A(m)$ otherwise,
- $m \vdash_{A_\perp} n$ iff $m = n = q$ or $m = q, n = a$ or $m = a, n \vdash_A n$ or $m \neq n, m \vdash_A n$.

Définition 1.3.26 (Stratégie)

Une stratégie σ sur une arène A est un ensemble de parties clos par préfixe pair de parties paires sur A, tel que

$$sab, sac \in \sigma \quad \Longrightarrow \quad sab = sac$$

Cette condition s'appelle déterminisme.

Définition 1.3.27 (Stratégies visibles, bien parenthésées, innocentes)

Une stratégie est visible si les coups du joueur pointent dans la vue.

Une stratégie est bien parenthésée si les réponses du joueur pointent toujours vers la dernière question de l'opposant dans la vue restée sans réponse.

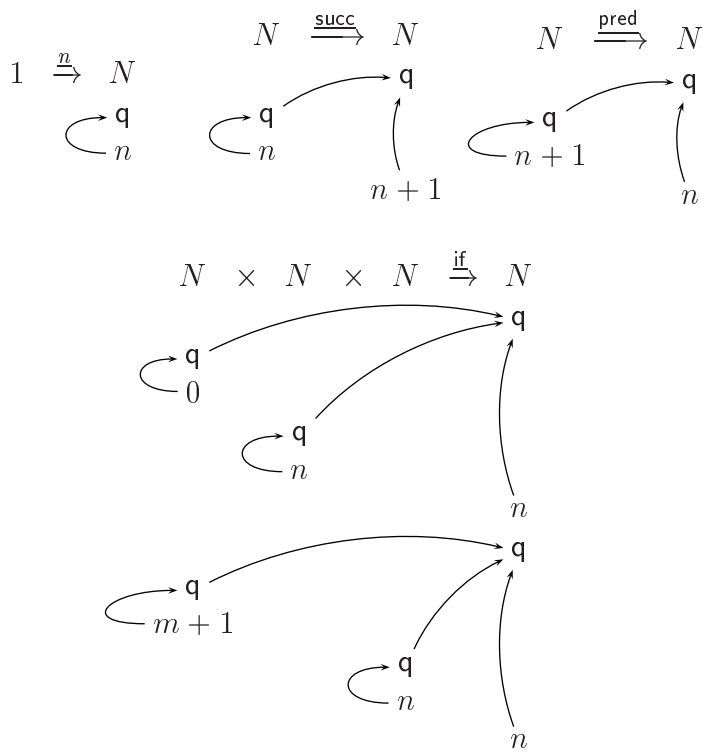
Une stratégie est filaire si toutes ses parties respectent le fil et si les coups du joueur ne dépendent que du fil : si $sab, t \in \sigma$ et $\lceil sa \rceil = \lceil ta \rceil$, alors il existe une unique facon d'étendre ta avec b telle que $\lceil sab \rceil = \lceil tab \rceil$.

Une stratégie visible est innocente si les coups du joueur ne dépendent que de la vue : si $sab, t \in \sigma$ et $\ulcorner sa \urcorner = \ulcorner ta \urcorner$, alors il existe une unique facon d'étendre ta avec b telle que $\ulcorner sab \urcorner = \ulcorner tab \urcorner$.

Théorème 1.3.28

Les arènes et les stratégies qui respectent le fil (resp. les stratégies innocentes) forment une catégorie cartésienne fermée. Les stratégies bien parenthésées forment des sous-catégories.

On représente graphiquement les parties par une suite de coups, et les pointeurs par des flèches entre les coups. La partie se lit de haut en bas, et les coups sont placés sous l'arène à laquelle ils appartiennent. Les stratégies pour compléter le modèle de PCF sont donc :



1.4 Monades

Les monades sont un outil catégorique très bien adapté à la représentation de comportements séquentiels des langages de programmation (voir par exemple [PJW93]). Si l'objet catégorique est connu depuis longtemps, son application à la sémantique est assez récent [Mog91, Mog89]. L'idée est qu'on distingue, dans la catégorie, les objets qui représentent les valeurs de ceux qui représentent les programmes : à chaque objet A pour les valeurs correspond un objet TA pour les programmes qui renvoient des valeurs dans A. On compose un programme M qui calcule une valeur de type A avec un

programme N qui utilise une valeur de A (contenue dans une variable x) pour produire une valeur de type B par la construction $\mathsf{let}\ x = M\ \mathsf{in}\ N$. Au contraire, à chaque valeur V correspond le programme trivial $\mathsf{val}\, V$. Ce langage s'appelle le lambda-calcul monadique.

Les constructions $\mathsf{let}\ x = -\ \mathsf{in}\ -$ et $\mathsf{val}\ -$ correspondent en termes catégoriques à la composition dans la catégorie de Kleisli et à la précomposition avec le morphisme η_A, respectivement.

Définition 1.4.1 (Monade)

Une monade sur une catégorie $\mathcal{C}$ est un triplet $\langle T, \eta, \mu\rangle$ où $T : \mathcal{C} \to \mathcal{C}$ est un foncteur, $\eta : 1 \to T$ et $\mu : T^2 \to T$ sont deux transformations naturelles telles que les diagrammes suivants commutent :

$$\begin{array}{ccccc} TA & \xrightarrow{T\eta_A} & TTA & \xleftarrow{\eta_{TA}} & TA \\ & \searrow^{\mathsf{id}_{TA}} & \downarrow{\scriptstyle A\mu} & \swarrow_{\mathsf{id}_{TA}} & \\ & & TA & & \end{array} \qquad \begin{array}{ccc} TTTA & \xrightarrow{T\mu_A} & TTA \\ \downarrow{\scriptstyle \mu_{TA}} & & \downarrow{\scriptstyle \mu_A} \\ TTA & \xrightarrow{\mu_A} & T_A \end{array}$$

Définition 1.4.2 (Catégorie de Kleisli)

Si $\mathcal{C}$ est une monade sur une catégorie $\mathcal{C}$, la catégorie de Kleisli $\mathcal{C}_T$ est définie par :

- $(\mathcal{C}_T)_0 = \mathcal{C}_0$,
- $\mathcal{C}_T[A, B] = \mathcal{C}[A, TB]$.

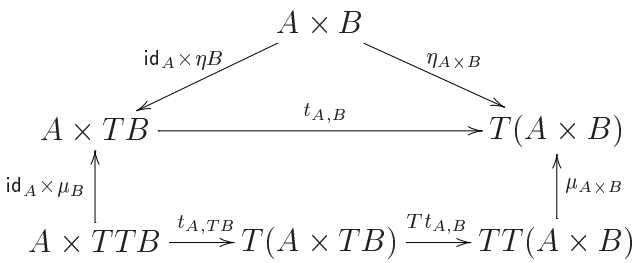
Pour $f \in \mathcal{C}[A, TB]$, on définit $f^* \in \mathcal{C}[TA, TB]$ par $f^* = \mu \circ Tf$. La composition dans $\mathcal{C}_T$ est définie par

$$f \bullet g = f^* \circ g = \mu \circ Tf \circ g$$

Une monade est dite forte s'il existe une transformation naturelle $t_{A,B} : (A \times TB) \to T(A \times B)$ telle que :

$$\begin{array}{ccc} 1 \times TA & \xrightarrow{t_{1,A}} & T(1 \times A) \\ & \searrow^{r_{TA}} & \downarrow{\scriptstyle Tr_A} \\ & & TA \end{array}$$

$$\begin{array}{ccccc} (A \times B) \times TC & & \xrightarrow{\qquad t_{A\times B,C} \qquad} & & T((A \times B) \times C) \\ \downarrow{\scriptstyle \alpha_{A,B,TC}} & & & & \downarrow{\scriptstyle T\alpha_{A,B,C}} \\ A \times (B \times TC) & \xrightarrow{A\times t_{B,C}} & A \times T(B \times C) & \xrightarrow{t_{A,B\times C}} & T(A \times (B \times C)) \end{array}$$



où r_A est simplement la seconde projection $\pi_2 : 1 \times A \to A$.

1.5 Équivalences contextuelles

Pour prouver qu'une optimisation est correcte, on voudra s'assurer que le programme proposé fait la même chose que le programme qu'on a donné au compilateur. Il reste à formaliser ce que "la même chose" veut dire. Pour un programme complet, la réponse est assez claire : les deux versions convergent vers la même valeur, ou divergent toutes les deux. Dans la pratique, le compilateur ne fait pas (ou peu) d'optimisations globales : il s'attache plutôt à améliorer localement le programme : partage de variables, inlining, déroulement de boucles, réordonnancement des instructions etc. Ces optimisations se font au niveau d'une phrase (d'un bloc pour les langages impératifs). Il s'agit donc, et c'est là que réside la difficulté du problème, de se convaincre que le fonctionnement global du programme n'est pas affecté, alors que le compilateur, quand il fait l'optimisation, n'en connaît qu'une petite partie. Comment savoir si la transformation locale aura ou non un impact sur le comportement du programme ?

Pour résoudre ce paradoxe, on n'autorisera que les transformations qui ne peuvent altérer aucun programme (et donc pas le programme qu'on compile) : il s'agit donc de prouver une propriété qui dit "quel que soit le programme autour de ce sous-terme, cette transformation n'aura pas d'incidence". Autrement dit, aucun programme qui entoure le sous terme considéré ne saura les distinguer. Dans ce cadre, l'observateur est donc le programme environnant : les possibilités d'observations sont intrinsèques au langage lui-même.

1.5.1 Définitions

Équivalence contextuelle syntaxique

On commence par formaliser la notion d'observation sur les programmes. Deux programmes M, M' sont équivalents s'ils convergent vers le même terme

N, ou s'ils divergent tous les deux. Pour modéliser la situation décrite plus haut (un sous-terme dans un programme), on définit la notion de contexte :

Définition 1.5.1 (Contexte)

Un contexte $C[-] : (\Gamma \vdash A) \Rightarrow (\Delta \vdash B)$ est un terme C avec un sous-terme distingué X tel que :
- $\Delta \vdash C : B$ et
- dans la preuve de $\Delta \vdash C : B$, X a le type $\Gamma' \vdash X : A$ avec $\Gamma' \supseteq \Gamma$.

On note $(\Gamma \vdash A)^\top$ le type $(\Gamma \vdash A) \Rightarrow (\vdash \iota)$.

Si $\Gamma \vdash M : A$, on note $C[M]$ la substitution avec capture de variables de X par M dans C.

Il faut noter que les contextes n'ont pas un type unique : si $C[-]$ a le type $(\Gamma \vdash A) \Rightarrow (\Delta \vdash B)$ et $\Gamma' \subseteq \Gamma$, alors $C[-]$ a aussi le type $(\Gamma' \vdash A) \Rightarrow (\Delta \vdash B)$.

Un terme $\Gamma \vdash M : A$ et un contexte $C[-] : (\Gamma \vdash A)^\top$ forment un programme $\vdash C[M] : \iota$. $C[M]$ est le programme global, M est la phrase qu'on veut optimiser, et $C[-]$ est la partie du programme inconnue lors de la compilation de M. L'ensemble des programmes dont M est un sous-terme est donc décrit par les $C[M]$, où C varie. On définit le préordre contextuel en quantifiant sur tous les contextes possibles.

Définition 1.5.2 (Préordre Contextuel)

Si M, N sont deux termes tels que $\Gamma \vdash M : A$ et $\Gamma \vdash N : A$, on note $\Gamma \vdash M \lesssim N : A$ si :

$$\forall C[-] : (\Gamma \vdash A)^\top, C[M] \Downarrow \implies C[N] \Downarrow$$

où $M \Downarrow$ signifie qu'il existe un terme M' tel que $M \Downarrow M'$ (on dit que M converge). On dit que M et N sont équivalents, et on note $\Gamma \vdash M \eqsim N : A$, si $\Gamma \vdash M \lesssim N : A$ et $\Gamma \vdash N \lesssim M : A$.

On écrira souvent $M \lesssim N$ et $M \eqsim N$, en omettant Γ et A.

Par définition, le préordre contextuel est une congruence :

Proposition 1.5.3

Si $C[-]$ est un contexte de type $(\Gamma \vdash A) \Rightarrow (\Delta \vdash B)$, et $\Gamma \vdash M \lesssim N : A$, alors $\Delta \vdash C[M] \lesssim C[N] : B$.

Démonstration : Si $C'[-]$ est un contexte de type $(\Delta \vdash B)^\top$, alors $C'' = C'[C[-]]$ est un contexte de type $(\Gamma \vdash A)^\top$. Si $C'[C[M]] = C''[M] \Downarrow$, alors $C'[C[N]] = C''[N] \Downarrow$ aussi. ■

La relation $\lesssim$ est évidemment réflexive et transitive, ce qui en fait un préordre. Ce n'est pas un ordre : le but étant de transformer des termes, il

est souhaitable que l'équivalence $\approx$ ne soit pas triviale. Dans le cas contraire, on ne pourrait absolument jamais échanger des termes, donc aucune optimisation ne serait possible.

En fait, pour les langages fonctionnels "purs" comme PCF, les seuls contextes intéressants sont les applications :

Lemme 1.5.4 (Lemme du contexte)

Soient P, P' deux termes fermés de type $A_1 \to \cdots \to A_n \to \iota$. $P \lesssim P'$ si et seulement si pour tous les n-uplets de termes fermés $M_1 : A_1, \ldots, M_n : A_n$ et tout entier m

$$P\, M_1 \ldots M_n \Downarrow m \implies P'\, M_1 \ldots M_n \Downarrow m$$

Démonstration : Ce lemme ne servant pas pour la suite de la thèse, nous avons choisi de faire une preuve sémantique, en utilisant des résultats prouvés plus loin.

Si $P \lesssim P'$, alors pour tout contexte $C[-]$, $C[P] \Downarrow m \implies C[P'] \Downarrow m$: si $M_1 \ldots M_n$ ont le bon type, $[-]\, M_1 \ldots M_n$ est bien un contexte pour P et P'.

Réciproquement, on suppose que $P\, M_1 \ldots M_n \Downarrow m \implies P'\, M_1 \ldots M_n \Downarrow m$. Il existe un modèle extensionnel avec la propriété de définissabilité (l'écrasement extensionnel du modèle de jeux par exemple) : comme tous les éléments compacts sont définissables, et comme le modèle de jeux est algébrique, $[\![P]\!]$ et $[\![P']\!]$ sont caractérisées (en extension) par leurs valeurs en les $[\![Q]\!]$. Par hypothèse, et par correction et adéquation de la sémantique, $[\![P]\!]$ est plus défini que $[\![Q]\!]$, c'est-à-dire $[\![P]\!] \sqsubseteq [\![Q]\!]$, et donc $[\![P]\!] \lesssim [\![Q]\!]$. Par la proposition 1.5.8, on obtient que $P \lesssim Q$. ■

Équivalence contextuelle sémantique

Un modèle définit une notion d'observation. On transpose les définitions syntaxiques dans le monde dénotationnel :

Définition 1.5.5 (Contexte)

Dans un modèle $\mathcal{M}$, un contexte de type $(\Gamma \vdash A)^\top$ est un morphisme $\alpha : ([\![\Gamma]\!] \Rightarrow [\![A]\!]) \to [\![\iota]\!]$. La mise en contexte de $f : [\![\Gamma]\!] \to [\![A]\!]$ est $\Lambda f; \alpha : 1 \to [\![\iota]\!]$.

Définition 1.5.6 (Préordre Contextuel)

Si $f, g : [\![\Gamma]\!] \to [\![A]\!]$ sont deux dénotations, on note $f \lesssim g$ si :

$$\forall \alpha : ([\![\Gamma]\!] \Rightarrow [\![A]\!]) \to [\![\iota]\!],\ \Lambda f; \alpha \sqsubseteq \Lambda g; \alpha$$

On note $\eqsim$ l'équivalence de ce préordre.

Lemme 1.5.7

Soit $\Gamma = x_1 : A_1 \dots x_n : A_n$ un environnement de typage. Soit M un terme tel que $\Gamma \vdash M : B$, et $C[-] : (\Gamma \to B)^\perp$ un contexte. Si z est une variable non liée dans $C[-]$, on a :

$$[\![\vdash C[M] : \iota]\!] = [\![z : \bar{A} \to B \vdash C[z\ \bar{x}] : \iota]\!] \circ [\![\vdash \lambda\bar{x}.M : \bar{A} \to B]\!]$$

où $\bar{A} \to B = A_1 \to \cdots \to A_n \to B$ et $\lambda\bar{x}.M = \lambda x_1 \dots \lambda x_n.M$.

Démonstration : Il faut en fait prouver par induction sur $C[-] : (\Gamma \to B) \Rightarrow (\Delta \to B')$ que

$$[\![\Delta \vdash C[M] : B']\!] = [\![\Delta, z : \bar{A} \to B \vdash C[z\ \bar{x}]]\!] \circ [\![\vdash \lambda\bar{x}.M : \bar{A} \to B]\!]$$

L'idée est simple : dans l'arbre de preuve de $\Delta \vdash C[M] : B'$, en remplaçant M par $z\ x_1 \dots x_n$ et en ajoutant $z : A_1 \to \cdots \to A_n \to B$ à tous les environnements de typage, on obtient un arbre de preuve de $\Delta, z : A_1 \to \cdots \to A_n \to B \vdash C[z\ x_1 \dots x_n] : B'$. Il est alors facile de vérifier que

$$\begin{aligned}[\![\Delta, z : \bar{A} \to B \vdash C[z\ \bar{x}] : B']\!] \circ \langle [\![\vdash \lambda\bar{x}.M : \bar{A} \to B]\!], \mathsf{id}_{[\![\Delta]\!]} \rangle \\ = [\![\Delta \vdash C[(\lambda\bar{x}.M)\ \bar{x}] : B']\!] \\ = [\![\Delta \vdash C[M] : B']\!]\end{aligned}$$

■

La proposition suivante montre que les modèles standards peuvent être utilisés pour prouver des équivalences contextuelles syntaxiques. Plus précisément, l'équivalence contextuelle dénotationnelle est plus forte que l'équivalence syntaxique :

Proposition 1.5.8

Soient M_1, M_2 deux termes tels que $\Gamma \vdash M_i : A$. Si $[\![\Gamma \vdash M_1 : A]\!] \lesssim [\![\Gamma \vdash M_2 : A]\!]$, dans un modèle standard, alors $\Gamma \vdash M_1 \lesssim M_2 : A$.

Démonstration : Supposons $[\![\Gamma \vdash M : A]\!] \lesssim [\![\Gamma \vdash M : A]\!]$. Soit $C : (\Gamma \vdash A)^\top$ tel que $C[M_1] \Downarrow V$. On écrit $\Gamma = x_1 : A_1 \dots x_n : A_n$. En appliquant le lemme 1.5.7, on obtient, pour $i = 1, 2$,

$$[\![C[M_i]]\!] = [\![z : \bar{A} \to B \vdash C[z\ \bar{x} : \iota]\!] \circ [\![\vdash \lambda\bar{x}.M_i : \bar{A} \to B]\!]$$

où $\bar{A} \to B = A_1 \to \cdots \to A_n \to A$ et $\lambda\bar{x}.M = \lambda x_1 \dots \lambda x_n.M$. On note $\alpha = [\![x : \bar{A} \to B \vdash C[x] : \iota]\!]$. Par correction du modèle,

$$[\![\vdash V : \iota]\!] = [\![C[M_i]]\!]$$

Par définition de l'observation dans le modèle,

$$[\![C[M_2]]\!] = \alpha; [\![\vdash \lambda\bar{y}.M_2 : \bar{A} \to B]\!] \sqsupseteq \alpha; [\![\vdash \lambda\bar{y}.M_1 : \bar{A} \to B]\!] = [\![\vdash V : \iota]\!]$$

Comme le modèle est standard, $[\![\iota]\!]$ est un cpo plat, donc $[\![\vdash V : \iota]\!]$, qui est maximal (par correction de la sémantique et définition d'un modèle standard, ça ne peut pas être $\bot$), ce qui donne $[\![C[M_2]]\!] = [\![\vdash V : \iota]\!]$. Par adéquation, on obtient bien :

$$C[M_2] \Downarrow V$$

■

En général, la réciproque n'est pas vraie (cf. exemple plus bas). Un bon modèle permettra de prouver plus d'équivalences, par exemple toutes les équivalences entre termes du premier ordre. Dans cette optique, les meilleurs modèles seront ceux qui vérifient une réciproque complète.

Définition 1.5.9 (Complète adéquation (Full abstraction))

On dit qu'un modèle est complètement adéquat si pour tous les termes M, N, $M \lesssim N$ si et seulement si $[\![M]\!] \lesssim [\![N]\!]$.

Milner [Mil77] a étudié la complète adéquation, et a prouvé qu'il existait toujours un modèle complètement adéquat (le modèle de termes quotienté par l'équivalence observationnelle) et que tous les autres modèles complètement adéquats lui étaient isomorphes. Pour avoir une idée des méthodes employées en sémantique dénotationnelle pour étudier l'adéquation complète avant la sémantique des jeux, on pourra consulter [BCL85].

Un modèle n'est pas complètement adéquat si les dénotations des contextes séparent des termes équivalents, i.e. elles font des observations impossibles dans le langage. A contrario, si toutes les dénotations correspondent à un terme, les observations qu'elles peuvent faire sont autorisées, et le modèle est complet.

Définition 1.5.10 (Définissabilité)

Un élément $f \in [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to [\![B]\!]$ est définissable s'il existe un terme M tel que $x_1 : A_1 \ldots x_n : A_n \vdash M : B$ et

$$f = [\![x_1 : A_1 \ldots x_n : A_n \vdash M : B]\!]$$

Proposition 1.5.11

Si tous les éléments du modèle sont définissables, alors le modèle est complètement adéquat.

Démonstration : Supposons $[\![M]\!] \not\lesssim [\![N]\!]$: il existe un contexte $\alpha \in [\![\bar{A} \to B]\!] \to [\![\iota]\!]$ tel que $[\![M]\!]; \alpha \neq \bot$ et $[\![N]\!]; \alpha = \bot$. Par hypothèse, il existe un terme C tel que $z : \bar{A} \to B \vdash C : \iota$ et $[\![C]\!] = \alpha$. On a $[\![(\lambda z.C)M]\!] = [\![M]\!]; [\![C]\!] \neq \bot$ donc $(\lambda z.C)M \Downarrow$ mais $[\![(\lambda z.C)N]\!] = [\![N]\!]; [\![C]\!] = \bot$ donc $(\lambda z.C)N \not\Downarrow$, ce qui prouve que $M \not\lesssim N$. ■

En général, le contexte α n'utilise qu'une partie finie de $[\![M]\!]$ pour converger, donc on peut le supposer compact. Il suffit alors de supposer que tous les éléments compacts sont définissables pour obtenir la complétude : c'est le cas par exemple de la sémantique des jeux pour PCF, et du modèle des algorithmes séquentiels pour SPCF (Sequential PCF, ou PCF avec des opérateurs de contrôle catch) [CF92].

Proposition 1.5.12

Si tous les éléments compacts d'un modèle de Scott sont définissables, alors il est complètement adéquat.

Démonstration : On reprend les définitions de la preuve précédente. Soit K l'ensemble des compacts dominés par $[\![M]\!]$: comme le domaine est algébrique, on a $[\![M]\!] = \bigsqcup K$. D'autre part, comme α est continu, $[\![M]\!]; \alpha = (\bigsqcup_{k \in K} k); \alpha = \bigsqcup_{k \in K} (k; \alpha)$. $[\![M]\!]; \alpha \neq \bot$, donc il existe $k \in K$ tel que $[\![M]\!]; k \neq \bot$. Par hypothèse, il existe C tel que $k = [\![x : A \vdash C]\!]$. On a donc

$$[\![\vdash (\lambda x.C)M]\!] = [\![M]\!]; [\![x : A \vdash C]\!] = [\![M]\!]; k \neq \bot$$

donc $C[M] \Downarrow$. D'autre part

$$[\![\vdash (\lambda x.C)N]\!] = [\![N]\!]; [\![x : A \vdash C]\!] = [\![N]\!]; k \sqsubseteq [\![N]\!]; \alpha = \bot$$

donc $C[N] \not\Downarrow$, ce qui prouve que $M \not\lesssim N$. ■

1.5.2 Cas du modèle de jeux

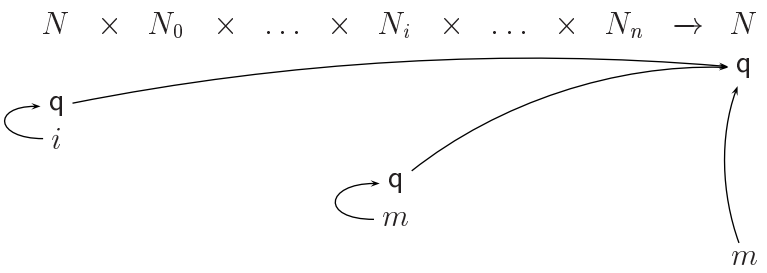
On va prouver ici que le modèle de jeux est complètement adéquat, grâce à la propriété de définissabilité : tous les compacts sont définissables. En fait, ce n'est pas tout à fait exact : la sémantique de PCF ne permet pas de mémoriser une valeur testée, ce qui fait qu'une séquence de tests sur le même terme repète le calcul. Il suffit, pour obtenir la propriété de définissabilité, d'ajouter une construction case au langage, qui permet de regrouper plusieurs tests en une seule opération, similaire à la construction switch...case du langage C.

Définition 1.5.13 (PCF+**case**)

PCF+case est l'extension de PCF obtenue en ajoutant des constantes

$\mathsf{case}_n : N^n \to N$ pour chaque $n \geq 0$ ($N^n \to N$ est $N \to \cdots \to N \to N$ avec $n+1$ fois N), avec $\mathsf{eval}_\delta(\mathsf{case}_n, j) = \pi_j^n$ si $j \leq n$ où $\pi_j^n = \lambda x_0 \lambda x_1 \ldots \lambda x_n . x_j$.

On interprète case_n par les stratégies avec les parties maximales suivantes, pour $0 \leq i \leq n$ et $m \leq 0$:



Proposition 1.5.14 (Définissabilité)

Si $\sigma : [\![\Gamma]\!] \to [\![A]\!]$ est une stratégie innocente et bien parenthésée compacte, alors il existe un terme P de PCF+case tel que $[\![\Gamma \vdash P : A]\!] = \sigma$.

Démonstration : Cf. [Har99] ■

Grâce au théorème 1.5.12, on peut donc prouver que le modèle de jeux de PCF+case est complètement adéquat. On obtient la complétude pour PCF en traduisant les contextes de PCF+case dans PCF : ils sont identiques, à ceci près que les case sont interprétés par des tests binaires.

La traduction $-^*$ de PCF+case dans PCF laisse tous les constructeurs invariants, à l'exception des case_n :

$$
\begin{array}{rcl}
(\mathsf{case}_n P Q_0 \ldots Q_n)^* & = & \mathsf{if}\ P^* \\
& & \mathsf{then}\ Q_0^* \\
& & \mathsf{else\ if\ pred}\ P^* \\
& & \quad \mathsf{then}\ Q_1^* \\
& & \quad \mathsf{else}\ \ldots \\
& & \quad\quad \mathsf{if\ pred}^n\ P^* \\
& & \quad\quad \mathsf{then}\ Q_n^* \\
& & \quad\quad \mathsf{else}\ \Omega \\
& & \quad\quad \mathsf{fi} \\
& & \quad\quad \ldots \\
& & \quad \mathsf{fi} \\
& & \mathsf{fi}
\end{array}
$$

On va maintenant vérifier que le comportement des termes traduits est le même que celui des termes d'origine : la seule différence est que certains sous-termes sont calculés plusieurs fois au lieu d'une, mais comme la réduction est

déterministe, le résultat est à chaque fois le même, et donc le résultat final est inchangé.

Proposition 1.5.15

> $P \Downarrow n$ dans PCF+case si et seulement si $P^* \Downarrow n$ dans PCF.

Démonstration : Comme la traduction $-^*$ est l'identité sur tous les constructeurs à l'exception de case_n, il suffit de prouver que $\mathsf{case}_n P Q_0 \dots Q_n \Downarrow m$ si et seulement si

$$\mathsf{if}\ P^*\ \mathsf{then}\ Q_0^*\ \mathsf{else}\ \dots \mathsf{if}\ \mathsf{pred}^n\ P^*\ \mathsf{then}\ Q_n^*\ \mathsf{else}\ \Omega\ \mathsf{fi} \dots\ \mathsf{fi} \Downarrow m$$

Si $\mathsf{case}_n P Q_0 \dots Q_n \Downarrow m$, il existe i tel que $P \Downarrow i$ et $Q_i \Downarrow m$. Alors, par hypothèse d'induction, $P^* \Downarrow i$, ce qui nous donne :

$$\dfrac{P^* \Downarrow i \qquad \dfrac{\mathsf{pred}\ P^* \Downarrow i-1 \qquad \dfrac{\mathsf{pred}^{i-1}\ P^* \Downarrow 1 \qquad \dfrac{\mathsf{pred}^i\ P^* \Downarrow 0 \qquad Q_i^* \Downarrow m}{\mathsf{if}\ \mathsf{pred}^i\ P^*\ \mathsf{then}\ Q_i^*\ \mathsf{else}\ \dots\ \mathsf{fi} \Downarrow m}}{\vdots}}{\mathsf{if}\ \mathsf{pred}^2\ P^*\ \mathsf{then}\ Q_2^*\ \mathsf{else}\ \dots\ \mathsf{fi} \Downarrow m}}{\mathsf{if}\ \mathsf{pred}\ P^*\ \mathsf{then}\ Q_1^*\ \mathsf{else}\ \dots\ \mathsf{fi} \Downarrow m}}{\mathsf{if}\ P^*\ \mathsf{then}\ Q_0^*\ \mathsf{else}\ \dots\ \mathsf{fi} \Downarrow m}$$

A contrario, si $(\mathsf{case}_n P Q_0 \dots Q_n)^* \Downarrow m$, il faut que pour un certain $0 \leq i \leq n$:

- $P^* \Downarrow k_0$ avec $k_0 > 0$
- $\mathsf{pred}\ P^* \Downarrow k_1$ avec $k_1 > 0$
- $\dots$
- $\mathsf{pred}^i\ P^* \Downarrow 0$
- $Q_i \Downarrow m$

Il est alors facile de vérifier que $P^* \Downarrow i$, ce qui nous donne

$$\dfrac{P \Downarrow i \qquad Q_i \Downarrow m}{\mathsf{case}_n P Q_0 \dots Q_n \Downarrow m}$$

■

On peut donc conclure :

Théorème 1.5.16 (Adéquation complète)

> Les stratégies innocentes et bien parenthésées forment un modèle complètement adéquat de PCF.

Démonstration : Si $[\![P]\!] \not\subseteq [\![Q]\!]$, il existe α compact qui les sépare. Par définissabilité, il existe un contexte C de PCF+case tel que $[\![C]\!] = \alpha$. Par adéquation, $C[P]$ et $C[Q]$ ne convergent pas tous les deux vers la même valeur. Supposons par exemple que $C[P] \Downarrow m$ et $C[Q] \not\Downarrow$. On a alors $(C[P])^* \Downarrow m$ et

$(C[Q])^* \not\Downarrow$. Comme $-^*$ est l'identité sur les termes de PCF, $(C[X]) = C^*[X]$ pour tout terme X de PCF, donc le contexte de PCF C^* sépare bien P et Q : $P \not\lesssim Q$ dans PCF. ■

1.6 Relations logiques

Comme on l'a vu, la "précision" d'un modèle (au sens de l'adéquation complète) est reliée directement à l'absence d'éléments non définissables. Les relations logiques, introduites en sémantique par Plotkin [Plo73], fournissent un moyen pratique de prouver qu'une fonction n'est pas définissable, ou (mais seulement jusqu'au deuxième ordre) de prouver qu'elle l'est.

1.6.1 Définition et propriétés

Si C est une catégorie cartésienne fermée, on notera, comme dans [AC98], $\underline{A} = C[1, A]$. Dans une catégorie cartésienne fermée, $C[A, B]$ est isomorphe à $C[1, A \to B]$.

Définition 1.6.1 (Relation logique)

Soient $\mathcal{M}_1 \dots \mathcal{M}_n$ des modèles. Une relation logique est une famille de relations

$$\mathcal{R}_A \subseteq \underline{[\![A]\!]^{\mathcal{M}_1}} \times \cdots \times \underline{[\![A]\!]^{\mathcal{M}_n}}$$

donnée pour le type de base et définie par induction aux autres types par

$$(f_1, \dots, f_n) \in \mathcal{R}_{A\to B} \iff \forall (x_1, \dots, x_n) \in \mathcal{R}_A, (f_1 x_1, \dots, f_n x_n) \in \mathcal{R}_B$$

Si $\mathcal{M}_1 = \cdots = \mathcal{M}_n$, un élément $f \in [\![A]\!]$ est dit invariant par $\mathcal{R}$ si $(f, \dots, f) \in \mathcal{R}_A$.

Définition 1.6.2

Soit $\mathcal{R}$ une relation logique. On étend la définition aux environnements : si $\Gamma = x_1 : A_1 \dots x_k : A_k$, alors

$$\mathcal{R}_\Gamma = \{\rho_1 \dots \rho_n \mid \forall i, (\rho_1 x_i, \dots, \rho_n x_i) \in \mathcal{R}_{A_i}\}$$

Les relations sur les dénotations ouvertes sont alors définies par :

$$\mathcal{R}_{\Gamma\vdash A} = \{(f_1, \dots, f_n) \mid \forall (\rho_1, \dots, \rho_n) \in \mathcal{R}_\Gamma, (f_1\rho_1, \dots, f_n\rho_n) \in \mathcal{R}_A\}$$

Lemme 1.6.3 (Lemme fondamental des relations logiques)

Si les (dénotations des) constantes sont invariantes par $\mathcal{R}$, alors pour tout terme $\Gamma \vdash P : A$, $[\![\Gamma \vdash P : A]\!]$ est invariant par $\mathcal{R}_{\Gamma \vdash A}$. La relation est alors dite relation de séquentialité.

Démonstration : Par induction sur P. ■

Définition 1.6.4

Une relation $\mathcal{R}$ entre deux ensembles X et Y est dite :
- surjective si pour tout $y \in Y$ il existe $x \in X$ tel que $(x, y) \in \mathcal{R}$,
- fonctionnelle si pour tout $x \in X$, il existe au plus un y tel que $(x, y) \in \mathcal{R}$.

Définition 1.6.5 (Modèle extensionnel pour l'ordre)

Un modèle standard $\mathcal{M}$ est extensionnel pour l'ordre si la relation logique $\sqsubseteq$ définie au type de base par $x \sqsubseteq x'$ si $x = \bot$ ou $x = x'$ définit un ordre partiel complet à tous les types, et si tous les éléments sont continus.

Il faut en particulier que $\sqsubseteq$ soit d'une part réflexif, ce qui implique que tous les éléments sont monotones, et d'autre part antisymétrique, ce qui implique que le modèle est extensionnel (que la catégorie a assez de points).

1.6.2 Relations de Sieber

La définition suivante est due à Sieber [Sie92].

Définition 1.6.6 (Relation de Sieber)

On définit pour chaque domaine de base $K_\bot$, et chaques $X, Y \subseteq \{1 \dots n\}$ une relation $\mathcal{S}^n_{X,Y} \subseteq K^n_\bot$:

$$(a_1, \dots, a_n) \in \mathcal{S}^n_{X,Y} \iff (\exists i \in X, a_i = \bot) \vee (\forall i, j \in Y, a_i = a_j)$$

Une relation de Sieber est une relation logique définie comme une intersection de $\mathcal{S}^n_{X,Y}$ au type de base.

Exemple : La relation $S = \mathcal{S}^2_{\{1\},\{1,2\}}$ définit l'ordre extensionnel : au type de base, $(x, y) \in S$ si et seulement si $x = \bot$ ou $x = y$ (CPO plat), et aux types fonctionnels, $(f, g) \in S$ seulement si $fx \sqsubseteq gy$ pour tous $x \sqsubseteq y$.

Exemple : La relation $S = \mathcal{S}^2_{\emptyset,\{1,2\}}$ est la relation d'extensionalité : un élément est invariant si (jusqu'à l'ordre 2, si et seulement si) il n'utilise que le comportement extensionnel de ses arguments.

Si on définit les fonctions $\wedge_g$ et $\wedge_d$ par les termes de PCF finitaire :

$$\begin{aligned} \wedge_g &= [\![\mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ 1\ \mathsf{fi}]\!] \\ \wedge_d &= [\![\mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ 1\ \mathsf{fi}]\!] \end{aligned}$$

On peut définir le « goûteur de et » qui envoie $\wedge_g$ sur 0 et $\wedge_d$ sur 1, et les autres fonctions sur $\bot$. Elle est bien définie et croissante, mais utilise une information non extensionnelle sur son argument : elle n'est pas invariante par S, donc pas définissable en PCF.

Proposition 1.6.7

Pour PCF finitaire, une relation logique est une relation de séquentialité si et seulement si c'est une relation de Sieber.

Démonstration : Cf. [AC98], page 97. ■

Théorème 1.6.8

Soit t une fonction d'ordre au plus 2 d'un modèle de PCF finitaire. Alors t est définissable si et seulement si elle est invariante par toutes les relations de Sieber.

Démonstration : Cf. [Sie92]. ■

por n'est pas définissable : On définit la fonction ou parallèle, notée por : elle renvoie vrai si un de ses arguments est vrai, et faux si les deux sont faux. Plus formellement, por est une fonction de $N_\bot \times N_\bot$ dans $N_\bot$ définie par :

$$\mathsf{por}\, x\, y = \begin{cases} 0 & \text{si } x = 0 \text{ ou } y = 0 \\ 1 & \text{si } x = 1 \text{ et } y = 1 \\ \bot & \text{sinon} \end{cases}$$

La fonction por n'est pas définissable : elle n'est pas invariante par la relation $S = S^3_{\{1,2\},\{1,2,3\}}$. En effet :

por	por	por	
0	$\bot$	1	$\in S$
$\bot$	0	1	$\in S$
0	0	1	$\notin S$

1.6.3 Quotients

On vient de voir que les relations logiques sont un outil (incomplet) pour décider de la définissabilité dans un modèle. Comme la qualité d'un modèle est reliée à la présence d'éléments non définissables, il semble naturel d'utiliser les relations logiques pour éliminer du superflu et obtenir un meilleur modèle : on appelle cette opération quotient par une relation logique.

Écrasement extensionnel

Définition 1.6.9 (Écrasement extensionnel)

Si $\mathcal{M}$ est un modèle, on définit un modèle quotient $\mathcal{M}'$:
- $\mathcal{R}$ est la relation logique telle que $\mathcal{R}_\iota$ est l'identité de $[\![\iota]\!]$,
- $[\![A]\!]^{\mathcal{M}'}$ est l'ensemble des classes d'équivalence de $[\![A]\!]^{\mathcal{M}}$ sous la relation $\mathcal{R}_A$,
- $[\![\Gamma \vdash P : A]\!]^{\mathcal{M}'}$ est la classe de $[\![\Gamma \vdash M : A]\!]^{\mathcal{M}}$

Proposition 1.6.10

Pour les langages considérés jusqu'ici (fonctionnels purs), l'écrasement d'un modèle standard est bien défini, et c'est un modèle extensionnel standard.

Démonstration : Dans toute la preuve, on notera $\overline{x}$ la classe de x.

Premièrement, il est facile de vérifier que $\mathcal{R}_A$ est une relation d'équivalence partielle (i.e. transitive et symétrique, mais pas forcément réflexive) sur $[\![A]\!]^{\mathcal{M}}$. $[\![A]\!]^{\mathcal{M}'}$ est donc bien défini.

D'autre part, par le lemme 1.6.3, la dénotation d'un terme est invariante par $\mathcal{R}$ (qui est une relation de séquentialité), donc dans le quotient : les dénotations sont bien définies.

Il reste à vérifier qu'on a bien une catégorie cartésienne fermée, c'est-à-dire que l'application est bien définie : si $\overline{f} \in [\![A \to B]\!]^{\mathcal{M}'}$ et $\overline{x} \in [\![A]\!]^{\mathcal{M}'}$, on définit $\overline{f}(\overline{x})$ comme $\overline{f(x)}$. Si on avait choisi d'autres représentants g et y des classes de f et de x, on aurait $f\mathcal{R}g$ et $x\mathcal{R}y$, donc $f(x)\mathcal{R}g(y)$, c'est-à-dire $\overline{f(x)} = \overline{g(y)}$.

Le modèle est extensionnel : si $\overline{f}, \overline{g}$ sont deux fonctions telles que pour chaque $\overline{x}$, $\overline{f}(\overline{x}) = \overline{g}(\overline{x})$ alors, pour tous $x\mathcal{R}y$, on a $\overline{x} = \overline{y}$, ce qui donne

$$\overline{fx} = \overline{f}(\overline{x}) = \overline{g}(\overline{x}) = \overline{g}(\overline{y}) = \overline{g(y)}$$

c'est-à-dire $f(x)\mathcal{R}g(y)$: on a prouvé $f\mathcal{R}g$, i.e. $\overline{f} = \overline{g}$. ■

Proposition 1.6.11

On suppose que le langage, toujours fonctionnel pur, a un nombre fini de constantes (en particulier, ce théorème s'applique à PCF finitaire et à PCF unaire). Si $\mathcal{M}$ est un modèle standard alors son écrasement extensionnel $\mathcal{M}'$ est un modèle extensionnel standard pour l'ordre.

Démonstration : Comme tous les domaines sont finis, la condition de continuité est en fait triviale (les ensembles dirigés ont un maximum). On se contente

donc de vérifier, par induction sur leur type, que les fonctions sont toutes monotones. Au type ι, il n'y a rien à vérifier.

On suppose la propriété vraie aux type A et B. Soit $\xi \in [\![A \to B]\!]^{\mathcal{M}'}$. Si $\alpha \sqsubseteq \beta \in [\![A]\!]^{\mathcal{M}'}$, alors, par construction de $\mathcal{M}'$, $\xi \sqsubseteq \xi$, et par définition d'une relation logique, $\xi\alpha \sqsubseteq \xi\beta$, donc ξ est bien monotone. ■

Quotient par une relation logique

Définition 1.6.12 (Quotient par une relation logique)

Soit $\mathcal{R}$ une relation de séquentialité, et $\mathcal{M}$ un modèle standard. On définit l'ensemble X_A (pour chaque type A) comme l'ensemble des éléments de $[\![A]\!]^{\mathcal{M}}$ invariants par $\mathcal{R}_A$.

Le modèle quotient $\mathcal{M}_{/\mathcal{R}}$ est l'écrasement extensionnel de la famille X_A.

Proposition 1.6.13

Le quotient d'un modèle standard est un modèle extensionnel standard.

Démonstration : Par le lemme fondamental des relations logiques, pour tout terme typé $\Gamma \vdash P : A$, $[\![\Gamma \vdash P : A]\!]$ est invariant par $\mathcal{R}$, donc un élément de X_A : on définit la dénotation $[\![\Gamma \vdash P : A]\!]^{\mathcal{M}_{/\mathcal{R}}}$, comme la classe de $[\![\Gamma \vdash P : A]\!]^{\mathcal{M}}$. Comme $\mathcal{R}$ est une relation de séquentialité, elle préserve toutes les constantes, donc les ensembles de flèches de 1 vers $[\![\iota]\!]$ sont les mêmes dans $\mathcal{M}$ et $\mathcal{M}_{\mathcal{R}}$: si $\Gamma \vdash P : \iota$, alors $[\![\Gamma \vdash MA]\!]^{\mathcal{M}_{/\mathcal{R}}} = [\![\Gamma \vdash P : A]\!]^{\mathcal{M}}$. La correction et l'adéquation de $\mathcal{M}_{\mathcal{R}}$ sont donc directement héritées de celles de $\mathcal{M}$. Le modèle $\mathcal{M}_{\mathcal{R}}$ est bien sûr extensionnel, puisqu'il résulte d'un écrasement extensionnel. ■

Chapitre 2

Définissabilité relative et hiérarchies de modèles

2.1 Généralités

2.1.1 Définissabilité relative

Comme on l'a vu dans la section précédente, les modèles ne sont pas complètement adéquats s'il contiennent des éléments non définissables. La question naturelle est : que font ces éléments, ou encore quel est le type d'observation interdite qu'ils font sur le terme qu'on leur donne ? Pour répondre à cette question, il faut partir de contre-exemples à la complète adéquation, et trouver quelle est la différence entre les deux termes, invisible syntaxiquement, qui est observable par les contextes du modèle.

Par exemple, dans PCF finitaire, on définit les termes

$$M = \lambda x.\mathsf{if}\ x\ \mathsf{then}\ \mathrm{tt}\ \mathsf{else}\ \mathrm{tt}\ \mathsf{fi} \qquad N = \lambda x.\mathrm{tt}$$

Il est évident que $M \lesssim N$ (dans le modèle de Scott, $[\![M]\!] \lesssim [\![N]\!]$), mais dans le modèle stable $[\![M]\!] \not\lesssim [\![N]\!]$, et la fonction qui renvoie tt sur $[\![M]\!]$ et ff sur $[\![N]\!]$ est stable. La différence entre M et N est que M utilise son argument avant de renvoyer une valeur constante, alors que N renvoie directement une valeur. Le contexte qui les sépare est capable de "voir" si une fonction utilise son argument, ce qui n'est pas possible syntaxiquement : le seul moyen de le savoir est de lui donner $\bot$, mais dans ce cas elle va diverger, et le contexte ne reprendra jamais la main.

Une fois qu'on a compris quel était le problème, peut-on construire un "contexte universel", qui incarne ce comportement ? Si un tel contexte existe, alors, en ajoutant un opérateur dans la syntaxe, avec un comprtement qui

lui correspond, on obtient un langage étendu, pour lequel notre modèle est complètement adéquat (tous les contextes sont maintenant définissables). Reste à définir la notion précise de contexte universel : l'idée est qu'il permet de définir tous les autres contextes, dans le sens suivant :

Définition 2.1.1 (Définissabilité relative)

Soit $\mathcal{L}$ un langage, et $\mathcal{M}$ un modèle de $\mathcal{L}$. Si $f : 1 \to [\![A]\!]^{\mathcal{M}}$ et $g : A \to [\![B]\!]^{\mathcal{M}}$, on dit que g est définissable à partir de f pour $\mathcal{L}$ s'il existe un terme clos $M : A \to B$ tel que :

$$\mathsf{ev} \circ \langle [\![\vdash M : A \to B]\!], f \rangle = g$$

On note $g \preceq^{\mathcal{M}}_{\text{def}} f$. C'est le préordre de définissabilité relative. L'équivalence associée définit les classes de définissabilité dans $\mathcal{M}$.

Notre observateur universel sera n'importe quel élément de la plus grande classe de définissabilité, si elle existe : ils sont tous inter-définissables, et ils permettent de définir tous les éléments du modèle, donc tous les contextes.

Il sera souvent aussi intéressant d'étudier les autres degrés de définissabilité : il arrive souvent que les fonctions "universelles" soient trop complexes pour être ajoutées à un langage de programmation, alors que des versions plus faibles sont implémentables. On pourrait alors imaginer d'étendre véritablement le langage : le modèle du langage reste un modèle pour ce nouveau langage, et s'il n'est toujours pas complètement adéquat (puisque le degré maximal n'est pas définissable), il est meilleur qu'avant, puisqu'une partie des contextes sémantiques qui posaient problèmes ont maintenant un alter ego syntaxique. Ainsi, le plus grand degré du modèle des fonctions séquentiellement réalisables de Longley [Lon02] a une complexité exponentielle (cf. [Roy04]) : s'il n'est pas raisonnable de l'inclure comme opération dans le langage, des opérations plus faibles (comme le test de strictness de notre exemple) sont par contre envisageables.

La plupart des résultats antérieurs sur la définissabilité relative concernent les modèles de Scott : Sazonov et Trakhtenbrot [Saz76, Tra75], qui ont initié ces recherches, ont exhibé une partie de l'ordre de définissabilité. La preuve de Plotkin [Plo80] de la complète adéquation du modèle de Scott pour PCF avec des opérateurs pour le parallèlisme indique qu'il existe un plus grand élément pour la définissabilité dans ce modèle (et, en adaptant la preuve, que le ou parallèle représente le plus grand degré pour PCF finitaire). Stoughton [Sto94] a présenté un algorithme qui décide, pour n'importe quelles fonctions f, g de PCF finitaire d'ordre au plus 2, si $f \preceq_{\text{def}} g$, et qui donne, le cas échéant, un terme qui définit f avec g. Loader [Loa01] a prouvé que l'équivalence observationnelle dans PCF finitaire était indécidable à partir de l'ordre

3 : il en découle que la définissabilité relative est également indécidable à cet ordre. Dans le cas de PCF, les relations logiques paramétriques se sont avérées très utiles : ainsi, Jung et Tiuryn [JT93] ont utilisées les relations de Kripke pour caractériser la définissabilité, et cette idée a été reprise par O'Hearn et Riecke [OR95] pour construire un modèle complètement adéquat de PCF.

La définissabilité dans les modèles stables a aussi été étudiée : il faut citer les travaux de Paolini [Pao04], qui a prouvé que le modèle stable de PCF finitaire était complètement adéquat pour le langage étendu avec le test donné en exemple (une fonction utilise-t-elle son argument ?) et une version à 6 arguments de la fonction de Gustave. Ehrhard et Colson [CE94, Ehr99] ont étudié la définissabilité relative dans le modèle fortement stable, et Longley [Lon02] a décrit une fonction universelle (un plus grand élément pour la définissabilité relative) dans le modèle des fonctions séquentiellement réalisables, qui sont en fait isomorphes aux fonctions fortement stables.

On peut d'ores et déjà énoncer deux résultats très généraux :

Proposition 2.1.2

Les éléments définissables d'un modèle forment la classe de définissabilité la plus petite, qu'on note $\bot_{\mathrm{def}}$.

Lemme 2.1.3

Dans les modèles de PCF finitaire et de PCF, deux degrés de définissabilité ont toujours une borne supérieure.

Démonstration : On choisit des représentants $\alpha : A_1 \to \dots A_k \to \iota$ et $\beta : B_1 \to \dots B_m \to \iota$ des deux classes. On définit le terme

$$M = \lambda a.\lambda b.\lambda n.\lambda s_1 \dots s_k t_1 \dots t_m.\mathsf{if}\ n\ \mathsf{then}\ a\ s_1 \dots s_k\ \mathsf{else}\ b\ t_1 \dots t_m\ \mathsf{fi}$$

La classe de définissabilité de la fonction $\gamma = [\![M]\!]\ \alpha\ \beta$ est la borne supérieure recherchée : elle est manifestement plus petite que toute classe qui permet de définir α et β, et si on définit Z_A le terme constant égal à 0 de type A, alors

$$\alpha = \lambda s_1 \dots s_k.\gamma\ 0\ s_1 \dots s_k\ Z_{B_1} \dots Z_{B_m}$$
$$\beta = \lambda t_1 \dots t_m.\gamma\ 1\ Z_{A_1} \dots Z_{A_m}\ t_1 \dots t_m$$

■

Les résultats plus précis dépendent du modèle considéré.

2.1.2 Hiérarchies de modèles

Les degrés de définissabilité sont reliés à la hiérarchie des modèles : l'idée est qu'un modèle contient les fonctions définissables, et d'autres fonctions.

Évidemment, s'il contient une fonction non définissable, il contient aussi toutes les fonctions de degré inférieur. On pourrait donc imaginer qu'à chaque degré de définissabilité corresponde un modèle.

Au contraire, à partir d'un modèle $\mathcal{M}$ et d'une relation logique $\mathcal{R}$ qui prouve qu'un élément de $\mathcal{M}$ n'est pas définissable, on peut construire le modèle quotient $\mathcal{M}_{/\mathcal{R}}$, qui sera plus petit, et donc meilleur. L'étude des degrés de définissabilité devrait donc permettre une meilleure compréhension des mécanismes qui font qu'un modèle n'est pas complètement adéquat, et diviser le problème en morceaux, dont certains seront peut-être plus faciles à résoudre.

On s'attache d'abord à préciser la notion de hiérarchie de domaines. Tous les modèles sont isomorphes au type N. Un modèle plus petit qu'un autre aura moins de fonctions au type $N \to N$: les contraintes à respecter au type $(N \to N) \to N$ seront moins nombreuses, et il est donc possible qu'il contienne plus de fonctions à ce type. On voit, en comparant des modèles de fonctions, que la définition n'est pas évidente. Quand on va vouloir comparer des modèles de nature différente, le problème sera d'autant plus complexe. On définit donc, de façon abstraite :

Définition 2.1.4 (Comparaison de modèles standard)

> Si $\mathcal{M}, \mathcal{M}'$ sont deux modèles standard, on appelle $\mathcal{R}_{\mathcal{M},\mathcal{M}'}$ la relation logique définie par la diagonale aux types de base.
>
> On dit que $\mathcal{M}$ est plus grand que $\mathcal{M}'$ si cette relation définit une fonction partielle surjective. $\mathcal{M}$ et $\mathcal{N}$ sont dits isomorphes si $\mathcal{M}$ est plus petit que $\mathcal{N}$ et vice-versa.

Il faut noter que, parce que les relations logiques ne se composent pas en général, cette notion de "plus grand que" n'est pas transitive.

La proposition suivante montre que cette notion recoupe effectivement les possibilités d'observation des modèles.

Proposition 2.1.5

> Si $\mathcal{M}$ est plus grand que $\mathcal{N}$, alors $\mathcal{M}$ sépare plus de termes que $\mathcal{N}$.

Démonstration : Supposons que $P \not\approx_{\mathcal{N}} Q : A$: il existe $\alpha^{\mathcal{N}} \in [\![A \to \iota]\!]^{\mathcal{N}}$ tel que $\alpha^{\mathcal{N}} \circ [\![P]\!]^{\mathcal{N}} \neq \alpha^{\mathcal{N}} \circ [\![Q]\!]^{\mathcal{N}}$. Comme $\mathcal{R}_{\mathcal{M},\mathcal{N}}$ est surjective, il existe $\alpha^{\mathcal{M}}$ tel que $\alpha^{\mathcal{M}} \mathcal{R} \alpha^{\mathcal{N}}$. Par le lemme fondamental des relations logiques, $[\![P]\!]^{\mathcal{M}} \mathcal{R} [\![P]\!]^{\mathcal{N}}$ (resp. $[\![Q]\!]$) donc $\alpha^{\mathcal{M}} \circ [\![P]\!]^{\mathcal{M}} \mathcal{R} \alpha^{\mathcal{N}} \circ [\![P]\!]^{\mathcal{N}}$ (resp. $[\![Q]\!]$). Comme $\mathcal{R}$ est l'identité au type de base, $\alpha^{\mathcal{M}} \circ [\![P]\!]^{\mathcal{M}} = \alpha^{\mathcal{N}} \circ [\![P]\!]^{\mathcal{N}}$ (resp. $[\![Q]\!]$), donc $\alpha^{\mathcal{M}} \circ [\![P]\!]^{\mathcal{M}} \neq \alpha^{\mathcal{M}} \circ [\![Q]\!]^{\mathcal{M}}$. ■

Sans grande surprise, les modèles dont les éléments sont définissables sont les plus petits.

Proposition 2.1.6

Un modèle extensionnel avec la propriété de définissabilité est toujours plus petit qu'un autre modèle.

Démonstration : Soit $\mathcal{M}$ un modèle, et $\mathcal{N}$ un modèle extensionnel avec la propriété de définissabilité. On note $\mathcal{R} = \mathcal{R}_{\mathcal{M},\mathcal{N}}$.

Si $g \in [\![A]\!]^{\mathcal{N}}$, alors il existe un terme $t : A$ tel que $[\![t]\!]^{\mathcal{N}} = g$. Par le lemme fondamental des relations logiques, $[\![t]\!]^{\mathcal{M}} \mathcal{R}^A [\![t]\!]^{\mathcal{N}}$ donc $\mathcal{R}^A$ est surjective.

Prouvons maintenant que $\mathcal{R}$ est fonctionnelle. Au type de base, c'est évident. Supposons que $f \mathcal{R}_{A\to B} g$ et $f \mathcal{R}^{A\to B} g'$. Soit $y \in [\![A]\!]^{\mathcal{N}}$. Comme $\mathcal{R}^A$ est surjective, il existe $x \in [\![A]\!]^{\mathcal{M}}$ tel que $x \mathcal{R}^A y$. Alors, $f x \mathcal{R}^B g y$ et $f x \mathcal{R}^B g' y$. Comme $\mathcal{R}^B$ est fonctionnelle, $gy = g'y$. On obtient $g = g'$, donc $\mathcal{R}^{A\to B}$ est fonctionnelle. ■

On notera dans toute la suite de ce chapitre $\mathcal{E}$ pour le modèle de Scott.

Proposition 2.1.7

Si un modèle extensionnel pour l'ordre $\mathcal{M}$ est plus petit que le modèle de Scott $\mathcal{E}$, alors la relation $\mathcal{R}_{\mathcal{E},\mathcal{M}}$ (en tant que fonction partielle) est monotone et continue.

Démonstration : On prouve par induction sur le type les propriétés de $\mathcal{R}$, à savoir :

- si $\mathcal{R}(x)$ et $\mathcal{R}(y)$ sont définis, et si $x \sqsubseteq y$, alors $\mathcal{R}(x) \sqsubseteq \mathcal{R}(y)$,
- si X est dirigé, et si pour tout $x \in X$, $\mathcal{R}(x)$ est défini, alors $\mathcal{R}(\bigsqcup X)$ est défini et $\mathcal{R}(\bigsqcup X) = \bigsqcup_{x\in X} \mathcal{R}(x)$,

C'est évident au type de base. On suppose que ces propriétés sont vraies au type A et B.

On suppose que x et y de $[\![A \to B]\!]^{\mathcal{E}}$ sont dans le domaine de $\mathcal{R}_{A\to B}$ et $x \sqsubseteq y$. On se donne $a' \sqsubseteq b'$ dans $[\![A]\!]^{\mathcal{M}}$. Comme $\mathcal{R}_A$ est surjective, il existe $a \in [\![A]\!]^{\mathcal{E}}$ tel que $\mathcal{R}_A(a) = a'$. On a

$$\mathcal{R}(x)a' = \mathcal{R}(x)\mathcal{R}(a) = \mathcal{R}(x(a)) \sqsubseteq \mathcal{R}(y(a)) = \mathcal{R}(y)\mathcal{R}(a) = \mathcal{R}(y)a'$$

D'autre part, comme $\mathcal{M}$ est extensionnel pour l'ordre, $\mathcal{R}(y)$ est monotone, et donc $\mathcal{R}(y)a' \sqsubseteq \mathcal{R}(y)b'$. On obtient $\mathcal{R}(x)a' \sqsubseteq \mathcal{R}(y)b'$, ce qui prouve que $\mathcal{R}(x) \sqsubseteq \mathcal{R}(y)$.

Soit X un ensemble dirigé de $[\![A \to B]\!]^{\mathcal{E}}$, et $a' \in [\![A]\!]^{\mathcal{M}}$. On choisit a tels que $\mathcal{R}_A(a) = a'$. X est dirigé et on vient de prouver que $\mathcal{R}_{A\to B}$ est monotone, donc $\{\mathcal{R}_{A\to B}(x)\}_{x\in X}$ est également dirigé. D'autre part,

$$\mathcal{R}_B(xa) = \mathcal{R}_{A\to B}(x)\mathcal{R}_A(a) = \mathcal{R}_{A\to B}(x)a'$$

Par hypothèse d'induction, $\mathcal{R}_B$ est continue, donc

$$\mathcal{R}_B((\bigsqcup X)a) = \mathcal{R}_B(\bigsqcup_{x\in X}(xa)) = \bigsqcup_{x\in X} \mathcal{R}_B(xa) = \bigsqcup_{x\in X} (\mathcal{R}_{A\to B}(x)a') = (\bigsqcup_{x\in X} \mathcal{R}_{A\to B}(x))a'$$

ce qui prouve que $\mathcal{R}_{A\to B}(\bigsqcup X) = \bigsqcup_{x\in X} \mathcal{R}_{A\to B}(x)$). ■

Proposition 2.1.8

On suppose que que chaque A, $[\![A]\!]^{\mathcal{E}}$ est fini, et que toutes les bornes supérieures y existent. Si $\mathcal{M}$ est un modèle extensionnel pour l'ordre, alors $\mathcal{M}$ est plus petit que $\mathcal{E}$.

En fait, il existe une famille de fonctions totales, monotones et continues $\mathcal{R}_A^{-1}$ de $[\![A]\!]^{\mathcal{M}}$ vers $[\![A]\!]^{\mathcal{E}}$ telles que pour chaque type A :

- si $a \in [\![A]\!]^{\mathcal{M}}$, alors $\mathcal{R}_A(\mathcal{R}_A^{-1}(a))$ est défini et il est égal à a,
- si $a \in [\![A]\!]^{\mathcal{E}}$ est tel que $\mathcal{R}(a)$ est défini, alors $\mathcal{R}_A^{-1}(\mathcal{R}_A(a)) \sqsubseteq a$.

Démonstration : On prouve l'ensemble par une même induction. C'est vrai au type de base : $\mathcal{R}$ est l'identité. On suppose que $\mathcal{R}$ est fonctionnelle et surjective aux types A et B, et qu'il existe $\mathcal{R}_A^{-1}$, $\mathcal{R}_B^{-1}$ qui conviennent. On définit $\mathcal{R}_{A\to B}^{-1}(f)$ pour $f \in [\![A \to B]\!]^{\mathcal{M}}$ par

$$\mathcal{R}_{A\to B}^{-1}(f)(a) = \bigsqcup_{a'\in\alpha(a)} \mathcal{R}_B^{-1}(f(\mathcal{R}_A(a')))$$

où $\alpha(a)$ est l'ensemble des a' compacts tels que $a' \sqsubseteq a$ et $\mathcal{R}(a')$ est défini. $\mathcal{R}^{-1}$ est bien définie pour chaque f et a car toutes les bornes supérieures existent. Comme $\alpha(a)$ grossit quand a est plus grand, il est clair que $\mathcal{R}^{-1}(f)$ est monotone. Si D est un ensemble dirigé de $[\![A]\!]^{\mathcal{E}}$, alors $\mathcal{R}^{-1}(f)(D)$ est également dirigé ($\mathcal{R}^{-1}(f)$ est monotone), et pour tout compact a'

$$a' \sqsubseteq \bigsqcup D \iff \exists a \in D, a' \sqsubseteq a$$

par définition, ce qui nous donne que $\alpha(\bigsqcup D) = \bigsqcup_{a\in D} \alpha(a)$. On obtient donc

$$\bigsqcup_{a'\in\alpha(\bigsqcup D)} \mathcal{R}_B^{-1}(f \circ \mathcal{R}_A(a')) = \bigsqcup_{a\in D} \bigsqcup_{a'\in\alpha(a)} \mathcal{R}_B^{-1}(f \circ \mathcal{R}_A(a'))$$

ce qui prouve que $\mathcal{R}^{-1}(f)$ est continue.

Prouvons maintenant que $\mathcal{R}^{-1}$ est une relation logique. Par définition, et par continuité de $\mathcal{R}_B^{-1}$ de f et de $\mathcal{R}_A$:

$$\begin{array}{rcl} \mathcal{R}_{A\to B}^{-1}(f)(\mathcal{R}_A^{-1}(a)) & = & \bigsqcup_{a'\in\alpha(\mathcal{R}_A^{-1}(a))} \mathcal{R}_B^{-1}(f(\mathcal{R}_A(a'))) \\ & = & \mathcal{R}_B^{-1}(f(\mathcal{R}_A(\bigsqcup_{a'\in\alpha(\mathcal{R}_A^{-1}(a))} a')))) \end{array}$$

Comme le domaine est fini, $R^{-1}(a)$ est compact, donc $\alpha(R^{-1}(a))$ est dirigé et $\mathcal{R}_A^{-1}(a) = \bigsqcup_{a'\in\alpha(R^{-1}(a))} a'$, ce qui donne

$$\mathcal{R}_{A\to B}^{-1}(f)(\mathcal{R}_A^{-1}(a)) = \mathcal{R}_B^{-1}(f(\mathcal{R}_A(\mathcal{R}_A^{-1}(a))))$$

On conclut en appliquant l'hypothèse d'induction qui dit que $\mathcal{R}_A \circ \mathcal{R}_A^{-1}$ est l'identité :

$$\mathcal{R}_{A\to B}^{-1}(f)(\mathcal{R}_A^{-1}(a)) = \mathcal{R}_B^{-1}(f(a))$$

Prouvons maintenant que $\mathcal{R}_{A\to B}$ est fonctionnelle et surjective. Soit $f \in [\![A \to B]\!]^{\mathcal{M}}$. Si $a\mathcal{R}a'$, alors $a' = \mathcal{R}_A(a)$, ce qui donne

$$\mathcal{R}_B(\mathcal{R}_{A\to B}^{-1}(f)(a)) = \mathcal{R}_B(\mathcal{R}_B^{-1}(f \circ a')) = f \circ a'$$

c'est-à-dire $\mathcal{R}_{A\to B}^{-1}(f)(a)\ \mathcal{R}\ f \circ a$. On vient de prouver que $\mathcal{R}_{A\to B}^{-1}(f)\ \mathcal{R}\ f$: $\mathcal{R}_{A\to B}$ est surjective.

On suppose maintenant que $f\ \mathcal{R}_{A\to B}\ g$ et $f\ \mathcal{R}_{A\to B}\ g'$. Pour tout $a \in [\![A]\!]^{\mathcal{M}}$, on a $\mathcal{R}^{-1}(a)\ \mathcal{R}\ a$ ce qui donne

$$f(\mathcal{R}_A^{-1}(a)) \quad \mathcal{R}_B \quad g \circ a \qquad\qquad f(\mathcal{R}_A^{-1}(a)) \quad \mathcal{R}_B \quad g' \circ a$$

Comme $\mathcal{R}_B$ est fonctionnelle, on obtient $g \circ a = g' \circ a$. Comme $\mathcal{M}$ est extensionnel, $g = g'$, ce qui prouve que $\mathcal{R}_{A\to B}$ est fonctionnelle.

Il reste à vérifier que $\mathcal{R}^{-1}$ définit une rétraction. On calcule, pour chaque $a \in [\![A]\!]^{\mathcal{M}}$:

$$\begin{array}{rcl}
(\mathcal{R}_{A\to B}(\mathcal{R}_{A\to B}^{-1}(f))) \circ a & = & (\mathcal{R}_{A\to B}(\mathcal{R}_{A\to B}^{-1}(f))) \circ \mathcal{R}_A(\mathcal{R}_A^{-1}(a)) \\
& = & \mathcal{R}_B(\mathcal{R}_{A\to B}^{-1}(f)(\mathcal{R}_A^{-1}(a))) \\
& = & \mathcal{R}_B(\mathcal{R}_B^{-1}(f \circ \mathcal{R}_A(\mathcal{R}_A^{-1}(a)))) \\
& = & \mathcal{R}_B(\mathcal{R}_B^{-1}(f \circ a)) \\
& = & f \circ a
\end{array}$$

et au contraire, pour chaque $f \in [\![A \to B]\!]^{\mathcal{E}}$ tel que $\mathcal{R}_{A\to B}(f)$ existe, on a pour tout $a \in [\![A]\!]^{\mathcal{E}}$

$$\begin{array}{rcl}
(\mathcal{R}_{A\to B}^{-1}(\mathcal{R}_{A\to B}(f)))(a) & = & \bigsqcup_{a'\in\alpha(a)} \mathcal{R}_B^{-1}(\mathcal{R}_{A\to B}(f) \circ \mathcal{R}_A(a')) \\
& = & \bigsqcup_{a'\in\alpha(a)} \mathcal{R}_B^{-1}(\mathcal{R}_B(f(a'))) \\
& = & \bigsqcup_{a'\in\alpha(a)} f(a')) \\
& = & f(\bigsqcup \alpha(a)) \\
& \sqsubseteq & f(a)
\end{array}$$

■

2.2 Définissabilité dans le modèle de Scott de PCF unaire

Cette section reprend [BLP03], travail commun avec Antonio Bucciarelli et Vincent Padovani. Comme on l'a vu, la hiérarchie des degrés de définissabilité est liée à la hiérarchie des modèles : en quotientant pour éliminer certains

éléments non définissables, on obtient un plus petit modèle. On peut donc se demander si les degrés de définissabilité sont tous séparables dans ce sens : si $f \prec_{\text{def}} g$, existe-t-il un modèle extensionnel qui contient f mais pas g ?

Nous examinons le cas de PCF unaire. Loader [Loa98] a montré que l'équivalence observationnelle y est décidable, et Schmidt-Schauss [SS99] a défini un algorithme qui énumère les classes d'équivalence à tous les types, ce qui en fait un bon candidat pour s'attaquer à cette question. On connait au moins deux modèles extensionnels de PCF unaire : le modèle complètement adéquat, et le modèle de Scott, qui n'est pas complètement adéquat, car il contient la fonction por qui n'est pas définissable (à proprement parler, il s'agit dans le cas de PCF finitaire d'un opérateur de convergence parallèle plutôt que d'une opération sur des booléens, mais nous avons choisi de garder la notation du "ou parallèle" pour ne pas multiplier les symboles). Nous examinons en fait deux problèmes :

1. la hiérarchie des degrés est-elle triviale, c'est-à-dire por et $\bot_{\text{def}}$ sont-ils les seuls degrés ?
2. la hiérarchie des modèles extensionnels est-elle triviale, c'est-à-dire $\mathcal{E}$ et le modèle complètement adéquat sont-ils les seuls modèles ?

Une réponse positive à la première question implique une réponse positive à la deuxième. On va en fait prouver que la hiérarchie des modèles est effectivement triviale, mais qu'il existe des degrés intermédiaires. Ceci indique que, quand on quotiente par une relation logique qui élimine por, on élimine également toutes les autres fonctions non définissables, soit lors de la première phase (elles n'étaient pas invariantes) ou la seconde (l'écrasement extensionnel).

Dans toute cette partie, on parlera de trace dans les modèles de Scott : cette notion n'est a priori définie que pour les modèles stables, mais on peut l'étendre à toutes les fonctions monotones dont le codomaine est plat. On utilisera également les formes curryfiées des fonctions implicitement : ainsi, si f est une fonction de $[\![A_1]\!] \to \cdots \to [\![A_n]\!] \to [\![\mathsf{U}]\!]$, sa trace sera l'ensemble des n-uplets minimaux (pour l'ordre produit) $(x_1, \ldots, x_k)$ tels que

$$f\ x_1\ \ldots\ x_n = \top$$

2.2.1 Quelques degrés de définissabilité

Un premier résultat est qu'il existe un élément universel dans le modèle de Scott $\mathcal{E}$: il s'agit de l'évaluation en parallèle de deux termes, qui termine si un des deux termine.

Proposition 2.2.1

On définit $-\vee-$: $[\![\mathsf{U}]\!]\times[\![\mathsf{U}]\!]\to[\![\mathsf{U}]\!]$ par $x\vee y=\begin{cases}\top & \text{si } x=\top \text{ ou } y=\top\\ \bot & \text{si } x=\bot=y\end{cases}$
$\vee$ est le plus grand degré de définissabilité dans $\mathcal{E}$, on note $\top_{\text{def}}$ son degré.

Démonstration : Par induction sur le type A. Les deux éléments de $[\![\mathsf{U}]\!]$ sont définissables, on considère donc le type $A=A_1\to A_2\to\cdots\to A_n\to\mathsf{U}$. Soit $f\in[\![A]\!]$, de trace $\{\varphi_1,\ldots,\varphi_k\}\subseteq[\![A_1]\!]\times\cdots\times[\![A_n]\!]$.

Si $x\in[\![A_i]\!]$, la fonction $\lambda y.(x\sqsubseteq y)$ qui renvoie $\top$ si $x\sqsubseteq y$ et $\bot$ sinon est définissable : si $\{\xi_1\ldots\xi_l\}$ est la trace de x, alors par hypothèse d'induction, il existe des termes $X_1\ldots X_l$ qui définissent $\xi_1\ldots\xi_l$, et $x\sqsubseteq y=(y\ X_1)\wedge\cdots\wedge(y\ X_l)$.

Il est facile de vérifier que $f=\lambda y.(\varphi_1\sqsubseteq y)\vee\cdots\vee(\varphi_k\sqsubseteq y)$, ce qui conclut la preuve. ■

Au premier ordre, la hiérarchie de définissabilité relative est triviale :

Proposition 2.2.2

Si $f:[\![\mathsf{U}]\!]^n\to[\![\mathsf{U}]\!]$ n'est pas définissable, alors il existe $u_1\ldots u_n$ dans $\{x,y,\top,\bot\}$ tels que $[\![\lambda xy.fu_1\ldots u_n]\!]=\lambda xy.x\vee y$: tous les éléments non définissables du premier ordre ont la puissance de $\vee$.

Démonstration : Soit $f\in[\![\mathsf{U}^n\to\mathsf{U}]\!]$ une fonction non définissable. Il existe deux éléments distincts ϕ_1 et ϕ_2 dans la trace de f. On note ϕ_j^i le i-ème élément de ϕ_j.

On définit une suite X_i par

$$X_i=\begin{cases}\top & \text{si } \phi_1^i=\phi_2^j=\top\\ x & \text{si } \phi_1^i=\top \text{ et } \phi_2^j=\bot\\ y & \text{si } \phi_1^i=\bot \text{ et } \phi_2^j=\top\\ \bot & \text{si } \phi_1^i=\phi_2^j=\bot\end{cases}$$

Il est alors facile de vérifier que $[\![\lambda f.fX_1\ldots X_n]\!]\ f=\lambda xy.x\vee y$. ■

On va maintenant exhiber des degrés intermédiaires, non définissables, mais sans la puissance de l'évaluation parallèle complète. Il va donc falloir chercher dans les types d'ordre supérieur : on définit la fonction $\phi\in[\![(\mathsf{U}^3\to\mathsf{U})\to(\mathsf{U}^3\to\mathsf{U})]\!]^{\mathcal{E}}$ par

$$\phi f=\begin{cases}\lambda xyz.x\vee y\vee z & \text{si } f=\lambda xyz.x\vee y\\ f & \text{sinon}\end{cases}$$

Proposition 2.2.3

ϕ n'est pas définissable, mais n'a pas la puissance de $\vee$:

$$\bot_{\text{def}} \prec_{\text{def}} \phi \prec_{\text{def}} \mathsf{por}$$

Démonstration : On prouve que ϕ n'est pas invariante par la relation de Sieber $S = \mathcal{S}^3_{\{1,2\},\{1,2,3\}}$: au type de base tous les triplets sauf $(\top, \top, \bot)$ sont dans S. On note $f = \lambda xyz.x \wedge y$, $g = \lambda xyz.x \vee y$, et $g' = \lambda xyz.x \vee y \vee z$.

Il est facile de vérifier que $(f, g, g) \in S_{\mathsf{U}^3 \to \mathsf{U}}$. Au contraire $(f, g', g') \notin S_{\mathsf{U}^3 \to \mathsf{U}}$:

f	g'	g'	
$\top$	$\bot$	$\bot$	$\in S$
$\top$	$\bot$	$\bot$	$\in S$
$\bot$	$\top$	$\bot$	$\in S$
$\top$	$\top$	$\bot$	$\notin S$

Comme $\phi f = f$ et $\phi g = g'$, ϕ n'est pas invariant par S, donc pas définissable.

Pour prouver que $\phi \prec_{\text{def}} \mathsf{por}$, on prouve que $\mathsf{por} \not\preceq_{\text{def}} \phi$. Par induction sur (n, m), on montre qu'il n'y a pas de terme t en forme normale η-longue avec n occurrences libres de f et m occurrences de $\wedge$ tel que $[\![\lambda f xy.t]\!]\phi = \vee$:

- $[\![\lambda f xy.x]\!]\phi \neq \vee$ et $[\![\lambda f xy.y]\!]\phi \neq \vee$,
- si $t = t_1 \wedge t_2$, alors
  - soit $[\![\lambda f xy.t_1]\!]\phi\bot\bot = \top$ et $[\![\lambda f xy.t_2]\!]\phi = \vee$,
  - soit $[\![\lambda f xy.t_1]\!]\phi = \vee$ et $[\![\lambda f xy.t_2]\!]\phi\bot\bot = \top$,
- si $t = (f(\lambda z_1 z_2 z_3.v))u_1u_2u_3$ où un $[\![\lambda f xy.u_i]\!]\phi$ n'est pas définissable, alors celui-ci est le seul non définissable au type $\mathsf{U} \to \mathsf{U} \to \mathsf{U}$, i.e. $\vee$,
- si $t = (f(\lambda z_1 z_2 z_3.v))u_1u_2u_3$ où aucun u_i ne contient d'occurrence de f et $[\![\lambda f xyz_1z_2z_3.v]\!]\phi$ n'est pas définissable, par le lemme 2.2.2,

  $$[\![\lambda f xy.v[x\backslash w_1, y\backslash w_2, z_1\backslash w_3, z_2\backslash w_4, z_3\backslash w_5]]\!]\phi = \vee$$

  pour $w_1 \dots w_5$ bien choisis dans $\{\top, \bot, x, y\}$. Le terme en question contient $n - 1$ occurrences de f.
- si $t = fvu_1u_2u_3$ où ni v ni aucun u_i ne contiennent d'occurrences de f, alors pour tout $(a, b) \in \{\top, \bot\}^2$, si on note $v' = v[x\backslash a, y\backslash b]$ et $u'_i = u_i[x\backslash a, y\backslash b]$, on a :

$$\begin{aligned} [\![\lambda f xy.t]\!]\phi[\![a]\!][\![b]\!] &= \phi[\![v']\!][\![u'_1]\!][\![u'_2]\!][\![u'_3]\!] \\ &= [\![v']\!][\![u'_1]\!][\![u'_2]\!][\![u'_3]\!] \\ &= [\![v'u'_1u'_2u'_3]\!] \\ &= [\![\lambda xy.vu_1u_2u_3]\!][\![a]\!][\![b]\!] \end{aligned}$$

  En d'autres termes, $[\![\lambda f xy.vu_1u_2u_3]\!]\phi = \vee$, alors que le terme en forme normale $vu_1u_2u_3$ ne contient aucune occurrence de f.

■

En montant encore dans les types, on peut construire une fonction de degré incomparable : on définit $\psi \in [\![((\mathsf{U}^3 \to \mathsf{U}) \to (\mathsf{U}^3 \to \mathsf{U})) \to \mathsf{U}^2 \to \mathsf{U}]\!]^{\mathcal{E}}$ par :

$$\psi\varphi = \begin{cases} \lambda xy.\top & \text{si } \varphi \sqsupset \phi \\ \lambda xy.x \vee y & \text{si } \varphi = \phi \\ \lambda xy.\bot & \text{si } \varphi \not\sqsupseteq \phi \end{cases}$$

Il est facile de vérifier que ψ est bien monotone.

Proposition 2.2.4

Les degrés de définissabilité de ϕ et ψ sont incomparables.

Démonstration : D'abord, si $\phi \succeq \psi$, on aurait $\phi \succeq \vee$ (si $[\![M]\!]\phi = \psi$, alors $[\![M]\!]\phi\phi = \psi\phi = \mathsf{por}$), ce qui contredit la proposition précédente. Pour prouver que $\phi \not\succeq_{\text{def}} \psi$, on prouve que ψ est invariante par la relation $S = \mathcal{S}^3_{\{1,2\},\{1,2,3\}}$: comme ϕ n'est pas invariante pour S, le lemme fondamental des relations logiques nous permet de conclure qu'on ne peut effectivement pas la définir avec ψ.

Soient $\theta_1, \theta_2, \theta_3$, x_1, x_2, x_3 tels que y_1, y_2, y_3, si

ψ	ϕ	ϕ	
θ_1	θ_2	θ_2	
x_1	x_2	x_3	$\in S$
y_1	y_2	y_3	$\in S$
$\top$	$\top$	$\bot$	$\notin S$

On veut prouver que $(\theta_1, \theta_2, \theta_3) \notin S$. On a $\theta_1, \theta_2 \sqsupseteq \phi$ et $\theta_3 \not\sqsupseteq \phi$. On examine les deux cas possibles (qui ne sont pas mutuellement exclusifs) : $\theta_3(\lambda xyz.x \vee y) \sqsubseteq \lambda xyz.x \vee y \vee z$ et il existe f_0, x_0, y_0, z_0 tels que $\theta_3 f_0 x_0 y_0 z_0 = \bot$ et $f_0 x_0 y_0 z_0 = \top$. Si aucune de ces deux propositions n'est vérifiée, alors $\theta_3 \sqsupseteq \phi$.

Dans le premier cas, on définit $f = \lambda xyz.x \wedge y$ et $g = \lambda xyz.x \vee y$, comme précédemment, et alors

θ_1	θ_2	θ_3	
f	g	g	$\in S$
$\top$	$\bot$	$\bot$	$\in S$
$\top$	$\bot$	$\bot$	$\in S$
$\bot$	$\top$	$\bot$	$\in S$
$\top$	$\top$	$\bot$	$\notin S$

Dans le deuxième cas, on définit f' par

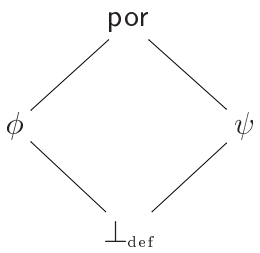
$$f'xyz = \begin{cases} \top & \text{si } x \sqsupseteq x_0,\ y \sqsupseteq y_0,\ z \sqsupseteq z_0 \\ \bot & \text{sinon} \end{cases}$$

f' est définissable, et $f' \sqsubseteq f$, donc $(f', f', f_0) \in S$. On a :

θ_1	θ_2	θ_3	
f'	f'	f_0	$\in S$
x_0	x_0	x_0	$\in S$
y_0	y_0	y_0	$\in S$
z_0	z_0	z_0	$\in S$
$\top$	$\top$	$\bot$	$\notin S$

■

On a donc décrit quatre degrés de définissabilité distincts :



2.2.2 Hiérarchie des modèles standard

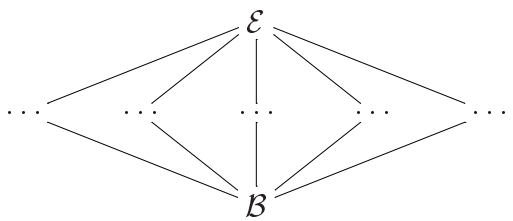
On va prouver ici qu'il n'y a que deux modèles extensionnels standard de PCF unaire. Dans cette partie, $\mathcal{B}$ est le modèle de bidomaines. Un premier résultat est que le modèle complètement adéquat est en fait le modèle de bidomaines.

Théorème 2.2.5

> Le modèle de bidomaines $\mathcal{B}$ de PCF unaire a la propriété de définissabilité, et est donc complètement adéquat.

La preuve est due à Laird [Lai03b].

Comme le domaine de base $\{\bot \sqsubseteq \top\}$ est un treillis, le théorème 2.1.8 complète une première image de la hiérarchie des modèles extensionnels de PCF unaire :



Il s'agit donc de prouver qu'il n'y a pas de modèle standard entre le modèle de bidomaines $\mathcal{B}$ et le modèle de Scott $\mathcal{E}$. On va en fait prouver que les fonctions qui ne sont pas dans $\mathcal{B}$ ont la puissance de $\vee$, qui est le degré maximal de définissabilité dans $\mathcal{E}$:

Lemme 2.2.6

On suppose que $\mathcal{M}$ est un modèle extensionnel standard pour l'ordre. Si les domaines de $\mathcal{M}$ et de $\mathcal{B}$ sont isomorphes pour les types plus petits que ou egaux à $A \to B$, et si $f : [\![A]\!]^{\mathcal{M}} \to [\![B]\!]^{\mathcal{M}}$, est $\sqsubseteq$-monotone et $\leq$-stable, alors $f \in [\![A \to B]\!]^{\mathcal{M}}$.

Démonstration : On écrit $R = R_{\mathcal{M},\mathcal{B}}$. Si $a, b \in [\![A]\!]^{\mathcal{M}}$, on définit $a \sqsubseteq b \iff R(a) \sqsubseteq R(b)$

Supposons que f est $\sqsubseteq$-continue and $\leq$-stable. Comme les domaines sont isomorphes, on peut définir $\hat{f}$ entre $[\![A]\!]^{\mathcal{B}}$ et $[\![B]\!]^{\mathcal{B}}$. $\hat{f}$ est également $\sqsubseteq$-monotone et $\leq$-stable : c'est un élément de $[\![A \to B]\!]^{\mathcal{B}}$.

Comme $R^{A\to B}$ est une bijection, on peut utiliser R^{-1}. Par définition de $R^{A\to B}$,

$$\forall x \in [\![A]\!]^{\mathcal{M}}, (R^{-1}(\hat{f})x, \hat{f}R(x)) \in R^B$$

Par définition de $\hat{f}$,

$$\forall x \in [\![A]\!]^{\mathcal{M}}, (fx, \hat{f}R(x)) \in R^B$$

Comme R^B est une bijection, $\forall x \in [\![A]\!]^{\mathcal{M}}, fx = R^{-1}(f)x$. On obtient $f = R^{-1}(\hat{f})$, ce qui signifie que $f \in [\![A \to B]\!]^{\mathcal{M}}$. ■

Proposition 2.2.7

Si un modèle extensionnel standard est strictement plus grand que le modèle de bidomaines, il contient la fonction $\vee$.

Démonstration : Soit $\mathcal{M}$ un modèle standard extensionnel strictement plus grand que $\mathcal{B}$. Soit C un type minimal tel que que $R_{\mathcal{M},\mathcal{B}}$ ne soit pas un isomorphisme. Il existe donc $\phi \in [\![C]\!]^{\mathcal{M}}$ qui n'a pas d'image par $R_{\mathcal{M},\mathcal{B}}$. C ne pouvant pas être le type de base, c'est un type fonctionnel $A \to B$. Les domaines pour A et B étant isomorphe, θ est une fonction entre $[\![A]\!]$ et $[\![B]\!]$ qui ne respecte pas les critères des bidomaines : étant monotone pour l'ordre extensionnel $\sqsubseteq$ ($\mathcal{M}$ est un modèle standard extensionnel), elle n'est soit pas monotone pour l'ordre stable $\leq$, soit pas stable.

Si θ n'est pas monotone pour $\leq$, il existe $f \leq f'$ dans $[\![A]\!]$ tels que $\theta f \not\leq \theta f'$. Comme l'ordre stable est un raffinement de l'ordre extensionnel dans la hiérarchie des types simples, $f \sqsubseteq f'$, et comme θ est monotone pour $\sqsubseteq$, $\theta f \sqsubseteq \theta f' : B$ ne peut pas être le type de base, c'est donc un type fonctionnel. θf et $\theta f'$ sont donc des fonctions stables : il existe $(x_1 \dots x_n)$ dans la trace de f qui n'est pas dans la trace de g. Comme $f \sqsubseteq g$, $gx_1 \dots x_n = \top$, donc il existe $x'_1 \dots x'_n \leq x_1 \dots x_n$ dans la trace de g. On définit les fonctions φ et ξ_i par :

- $\varphi\bot = f$ et $\varphi\top = f'$ et

– $\xi_i \bot = x'_i$ et $\xi_i \top = x_i$.

Elles vérifient les conditions pour être dans $\mathcal{B}$, et comme, à leurs types, les domaines sont isomorphes, elles sont aussi dans $\mathcal{M}$. La fonction

$$\vee \ = \ \lambda ab.(\theta(\varphi a))(\xi_1 b)\dots(\xi_n b)$$

est donc aussi dans $\mathcal{M}$.

Supposons maintenant que θ est $\leq$-monotone, mais qu'il existe f, f' bornés pour $\leq$ tels que $\theta(f \wedge f') \neq (\theta f) \wedge (\theta f')$. Comme θ est $\leq$-monotone, $\theta(f \wedge f') \sqsubseteq (\theta f) \wedge (\theta f')$: il existe $x_1 \dots x_n$ (n pouvant être 0) tels que

$$\theta(f \wedge f')x_1 \dots x_n = \bot$$

$$((\theta f) \wedge (\theta f'))x_1 \dots x_n = \top$$

On définit φ par :

– $\varphi \bot \bot = f \sqcap f'$,
– $\varphi \top \bot = f$,
– $\varphi \bot \top = f'$,
– $\varphi \top \top = g$.

où g est une borne supérieure de f, f'. φ est monotone pour $\sqsubseteq$, pour $\leq$ et elle est stable : elle est dans $\mathcal{B}$, et donc dans $\mathcal{M}$. Ainsi,

$$\vee \ = \ \underline{\lambda} ab.\theta(\varphi ab)x_1 ... x_n$$

est aussi dans $\mathcal{M}$. ■

Tous les éléments sont donc réunis pour prouver :

Théorème 2.2.8

Il n'existe que deux modèles extensionnels standard de PCF unaire : le modèle de bidomaines et le modèle de Scott.

Il faut noter qu'il existe une autre preuve de ce théorème, également due à Laird [Lai03b], qui repose sur l'algorithme de Schmidt-Schauss [SS99]. Notre preuve est plus générale, et s'applique en fait aussi au cas de PCF finitaire :

Théorème 2.2.9

On définit $\vee^-$, une version plus faible du ou parallèle, par

$$\vee^- xy = \begin{cases} \text{tt} & \text{si } x = \text{tt ou } y = \text{tt} \\ \bot & \text{sinon} \end{cases}$$

Tout modèle strictement plus grand que $\mathcal{B}$ contient $\vee^-$.

Par contre, dans le cas de PCF finitaire, le modèle de Scott n'est pas le plus grand des modèles extensionnels, car les domaines ne sont pas des treillis.

En fait, ce qui semble faire marcher ces preuves est la présence de "bornes supérieures", comme le confirment les travaux de Laird sur la bistabilité et les domaines localement booléens [Lai03a, Lai04].

2.3 Définissabilité dans le modèle de Scott de PCF finitaire

Cette section reprend [BL04], travail commun avec Antonio Bucciarelli. On a vu que le modèle des fonctions fortement stables était complet au premier ordre. Il semble donc assez naturel d'étudier la définissabilité relative au premier ordre en classant les fonctions selon leurs infractions à la stabilité forte. Une fonction croissante étant fortement stable si sa trace ne contient pas de sous-ensemble cohérent, on va caractériser les degrés de définissabilité à travers des hypergraphes : les sommets seront les points de la trace, et les hyperarcs les ensembles cohérents.

Ces travaux prolongent les résultats antérieurs de Bucciarelli et Malacaria [BM02, Buc97].

2.3.1 Hypergraphes et h-morphismes

Définition 2.3.1

Un hypergraphe coloré $H = \langle V_H, A_H, C_H \rangle$ est défini par :
- un ensemble fini V_H de sommets,
- un ensemble $A_H \subseteq \{A \subseteq V_H | \#A \geq 2\}$ d'arcs,
- un coloriage $C_H : V_H \to \{\mathrm{ff}, \mathrm{tt}\}$.

Définition 2.3.2

Soit $f : \mathsf{B}^n \to \mathsf{B}$ une fonction n-aire, de trace $\mathsf{tr}(f) = \{(v_1, b_1), \ldots, (v_k, b_k)\}$. L'hypergraphe H_f est défini par :
- $V_{H_f} = \{v_i\}$,
- A_{H_f} est l'ensemble des sous ensembles cohérents de V_{H_f} de cardinal au moins 2,
- $C_{H_f}(v_i) = b_i$

Il est facile de vérifier que les hypergraphes associés à une fonction monotone vérifient :

H1 : Si $\{x, y\} \in A_H$, alors $C_H(x) = C_H(y)$.

H2 : Si X_1, X_2 sont des arcs, et si $X_1 \cap X_2 \neq \emptyset$ alors $X_1 \cup X_2$ est aussi un arc.

Ces hypergraphes sont dits fonctionnels.

L'outil principal est le morphisme d'hypergraphe :

Définition 2.3.3 (***h***-morphisme)

Un h-morphisme d'un hypergraphe H vers un hypergraphe K est une fonction $m : V_H \to V_K$ entre les sommets telle que :
- pour tout $A \subseteq V_H$, si $A \in A_H$ alors $m(A) \in A_K$.
- pour tout $X \in A_H$, si $x, x' \in X$ et $C_H(x) \neq C_H(x')$ alors $C_K(m(x)) \neq C_K(m(x'))$.

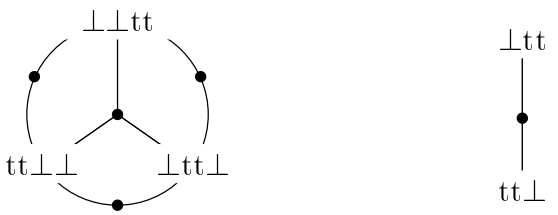
On cherche une fonction h qui envoie les points de la trace d'une fonction f vers les points de la trace d'une autre fonction g. h "adapte" g pour lui faire accepter la trace de f : elle transforme f en g. Comme h doit respecter la cohérence, elle est définissable, ce qui nous donnera une caractérisation de la définissabilité relative.

Plus précisément, Bucciarelli et Malacaria [BL04] ont prouvé que s'il existe un h-morphisme entre H_f et H_g, alors $f \leq_{\text{def}} g$ (nous ne rappelons pas la preuve ici, car la nouvelle preuve pour le cas généralisé est très similaire). La réciproque est fausse : soient $\mathsf{por}_3 : \mathsf{B}^3 \to \mathsf{B}$ et $\mathsf{por}_2 : \mathsf{B}^2 \to \mathsf{B}$ les fonctions définies par

$$\mathsf{por}_3(x,y,z) = \begin{cases} \text{tt} & \text{si } x,\ y, \text{ ou } z \text{ est tt} \\ \bot & \text{sinon} \end{cases}$$

$$\mathsf{por}_2(x,y) = \begin{cases} \text{tt} & \text{si } x \text{ ou } y \text{ est tt} \\ \bot & \text{sinon} \end{cases}$$

Les hypergraphes associés sont :

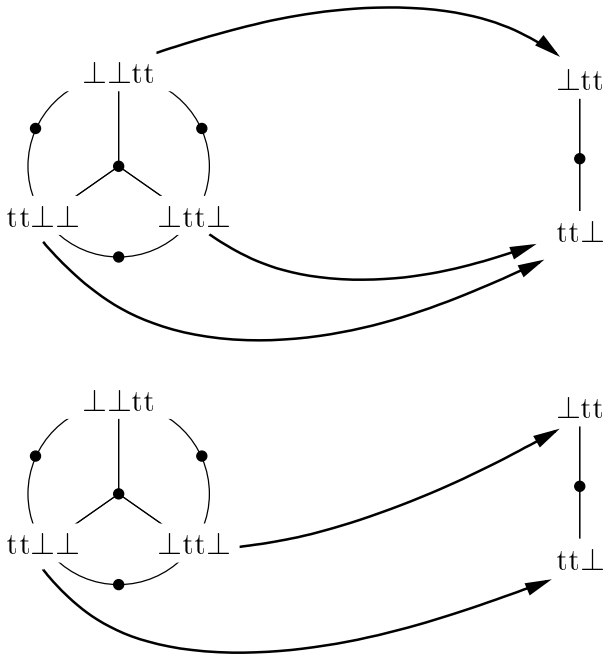


Il n'existe pas de h-morphisme $H_3 \to H_2$ (un des hyperarcs à deux sommets serait écrasé sur un point), et pourtant $\mathsf{por}_3 = [\![M]\!]\mathsf{por}_2$ avec

$$M = \lambda f\ \lambda x_1 x_2 x_3\ \mathsf{if}\ f(f(x_1,x_2),x_3)\ \mathsf{then}\ \text{tt}\ \mathsf{else}\ \bot\ \mathsf{fi}$$

2.3.2 Morphismes temporisés

En fait, il y a des appels imbriqués à f dans M. C'est nécessaire pour définir por_3 avec por_2, mais cela n'est pas modélisable par un h-morphisme : si on examine où M envoie chaque point de la trace de por_3, on constate que l'appel extérieur à f envoie $\bot tt\bot$ et $tt\bot\bot$ sur $tt\bot$ et $\bot\bot tt$ sur $\bot tt$, et que l'appel imbriqué envoie $\bot tt\bot$ sur $\bot tt$ et $tt\bot\bot$ sur $tt\bot$:



L'appel imbriqué à f "corrige" l'appel extérieur, qui écrasait un hyperarc sur un point, en séparant ces points. Pour modéliser les appels imbriqués, on définit donc la notion d'hypergraphe temporisé : il s'agit d'une suite (finie) de morphismes "simples" (qui peuvent écraser un hyperarc) qui vérifie cette propriété. Plus formellement, on définit d'abord la complétion d'un hypergraphe :

Définition 2.3.4

Si $H = \langle V_H, A_H, C_H \rangle$ est un hypergraphe fonctionnel, on définit sa complétion $\overline{H}$ par :
- $V_{\overline{H}} = V_H$ et $C_{\overline{H}} = C_H$,
- $A_{\overline{H}} = A_H \cup \{\{v\} \mid v \in V_H\}$.

Si H, K sont des hypergraphes, un h-morphisme de H dans $\overline{K}$ est dit non trivial si son image n'est pas un singleton.

Définition 2.3.5

Si $H = \langle V_H, A_H, C_H \rangle$ est un hypergraphe et $B \subseteq V_H$, on définit le sous hypergraphe $H\restriction_B$ par

- $V_{H\restriction_B} = B$ et
- $C_{H\restriction_B} = C_H\restriction_B$,
- $A_{H\restriction_B} = \{X \in A_H \mid X \subseteq B\}$.

On peut maintenant définir les morphismes temporisés, ainsi nommés puisqu'ils ajouent, par rapport aux morphismes simples présentés précédemment, une notion de séquence, même si, formellement, cela n'apparaît pas dans la définition :

Définition 2.3.6

Si H, K sont des hypergraphes fonctionnels, un morphisme temporisé α de H dans K est une famille α_X pour $X \in A_H$ de morphismes non triviaux de $H\restriction_X$ dans $\overline{K}$, avec la condition suivante (redondance) :

$$\forall X, Y \in A_H, \begin{cases} X \subseteq Y \\ \alpha_Y\restriction_X \text{ n'est pas trivial} \end{cases} \implies \alpha_X = \alpha_Y\restriction_X$$

où $\alpha_Y\restriction_X$ est la restriction de la fonction α_Y à l'ensemble X.

Cette définition est équivalente à l'intuition des suites de morphismes donnée plus haut. Si on a une suite $m_1 \dots m_k$ de morphismes, on peut définir une famille α_X par $\alpha_X = m_j\restriction_X$ pour le plus petit j tel que ce morphisme n'est pas trivial. Réciproquement, si on dispose d'une famille α_X, on veut construire une suite m_i. Si $A_1 \dots A_p$ sont les hyperarcs maximaux, on définit m_1 comme la réunion des α_{A_i} pour $1 \leq i \leq p$. On recommence avec les hyperarcs maximaux écrasés par m_1 pour obtenir m_2 etc. Comme il n'y a qu'un nombre fini d'hyperarcs, on obtient ainsi une suite finie qui convient. Les hypergraphes et les morphismes temporisés forment une catégorie, que l'on notera $\mathcal{TM}$.

Le lemme suivant sera utile à la preuve de correction :

Lemme 2.3.7

On peut restreindre un morphisme temporisé $\alpha : H \to K$ à un sous-ensemble $B \subset V_H$ par $(\alpha\restriction_B)_X = \alpha_X$ quand X est un hypergraphe de $H\restriction_B$. De plus, si α est un morphisme temporisé de H_f vers H_g, et que $\mathsf{tr}(f') \subseteq \mathsf{tr}(f)$, alors $\alpha\restriction_{\mathsf{tr}(f')}$ est un morphisme temporisé de $H_{f'}$ dans H_g.

2.3.3 Fonctions sous-séquentielles

Définition 2.3.8 (Fonction sous-séquentielle)

Une fonction $f : \mathsf{B}^n \to \mathsf{B}$ est sous-séquentielle s'il existe une fonction séquentielle (donc définissable) qui la majore pour l'ordre extensionnel.

Les fonctions sous-séquentielles ont des caractérisations en termes de graphes et de cohérence.

Proposition 2.3.9

Soit $f : \mathsf{B}^n \to N$ une fonction monotone. Les propositions suivantes sont équivalentes :

1. f est sous-séquentielle,
2. Pour tout $A \in \mathcal{C}(\mathsf{B}^n)$, $f(A) \in \mathcal{C}(\mathsf{B})$ (f préserve la cohérence linéaire),
3. Si $X \in A_{H_f}$, alors pour tout $x, y \in X$, $C_{H_f}(x) = C_{H_f}(y)$ (X est monochrome).

Si $A = \{v_1 \dots v_k\} \subseteq \mathsf{B}^n$, il existe en général plusieurs fonctions dont l'ensemble des points minimaux est exactement A. Par exemple, si les v_i sont deux à deux non bornés, il y en a 2^k. Le lemme suivant énonce que, parmi ces fonctions, celles qui sont sous-séquentielles ont le plus petit degré de définissabilité.

Lemme 2.3.10

Soient $f, g : \mathsf{B}^n \to \mathsf{B}$. Si g est sous-séquentielle et $\pi_1(\mathsf{tr}(f)) = \pi_1(\mathsf{tr}(g))$, alors $g \preceq_{\text{def}} f$.

On va prouver que s'il existe un morphisme temporisé $\alpha : H_f \to H_g$, alors $f \preceq_{\text{def}} g$. Le lemme suivant introduit la notion de tranche, qui sera centrale dans la preuve. Pour transformer $g : \mathsf{B}^n \to \mathsf{B}$ en $f : \mathsf{B}^m \to \mathsf{B}$, on va définir une fonction qui transforme les points minimaux de f dans B^m en points minimaux de g dans B^n : on représente cette fonction $\phi : \mathsf{B}^m \to \mathsf{B}^n$ comme n fonctions $\phi_1, \dots, \phi_n : \mathsf{B}^m \to \mathsf{B}$. Si ces fonctions sont définissables, on a déjà une fonction h qui termine si et seulement si f termine, à savoir :

$$h = \lambda \bar{x}.g(\phi_1 \bar{x}) \dots (\phi_n \bar{x})$$

Il ne restera alors qu'à accorder f et h quand elles convergent. Pour l'instant, on prouve que si les ϕ_i sont définies par un morphisme temporisé, elles sont alors sous-séquentielles.

Lemme 2.3.11

Soient $f : \mathsf{B}^m \to \mathsf{B}$, $g : \mathsf{B}^n \to \mathsf{B}$ des fonctions monotones et $\alpha : H_f \to H_g$ un morphisme temporisé. Pour chaque $X \in A_{H_f}$, $1 \leq i \leq n$, soit $f_i^X : \mathsf{B}^m \to \mathsf{B}$ la fonction de trace

$$\mathsf{tr}(f_i^X) = \{(v, \pi_i(\alpha_X(v))) \mid v \in X, \alpha_X(v)_i \neq \bot\}$$

Alors f_i^X est sous-séquentielle. On appelle f_i^X la $i^{\text{ème}}$ tranche de α_B.

Démonstration : Soit $Y \subseteq \mathsf{tr}(f_i^X)$ tel que $\pi_1(Y) \in \mathcal{C}(\mathsf{B}^m)$. Comme α est un morphisme temporisé, $\alpha_X(Y)$ est cohérent, donc $\pi_2(Y) \in \mathcal{C}(\mathsf{B}^m)$ également. On conclut par le lemme 2.3.9. ■

2.3.4 Correction

Nous avons maintenant tous les outils pour prouver la correction de la représentation par des hypergraphes :

Théorème 2.3.12

Soient $f : \mathsf{B}^l \to \mathsf{B}$, $g : \mathsf{B}^m \to \mathsf{B}$ des fonctions monotones telles que $\mathcal{TM}[H_f, H_g] \neq \emptyset$. Alors $f \preceq_{\text{def}} g$.

Démonstration : Soit $\alpha \in \mathcal{TM}[H_f, H_g]$. On prouve le théorème par induction sur k, le cardinal de $\mathsf{tr}(f)$. Si $k = 1$, f est séquentielle, donc définissable, ce qui donne $f \preceq_{\text{def}} g$. Si $k > 1$, on raisonne par cas sur la structure de H.

Supposons dans un premier temps que $V_{H_f} \notin A_{H_f}$: il existe $1 \leq i \leq l$ tel que $\pi_i(\pi_1(\mathsf{tr}(f))) = \{\mathrm{tt}, \mathrm{ff}\}$ (c'est l'index de séquentialité de f). Si on divise la trace de f selon la valeur de ce paramètre, on définit deux fonctions f_{tt} et f_{ff}.

Pour $\rho = \mathrm{tt}, \mathrm{ff}$, $\mathsf{tr}(f_\rho) \subset \mathsf{tr}(f)$. Par le lemme 2.3.7, il existe un morphisme temporisé de H_{f_ρ} dans H_g, et l'hypothèse d'induction nous donne que $H_{f_\rho} \preceq_{\text{def}} g$. Appelons M_ρ un terme qui définit f_ρ à partir de g. Il est alors facile de vérifier que

$$M = \lambda g \lambda \bar{x}.\mathsf{if}\ x_i\ \mathsf{then}\ M_{\mathrm{tt}} g \bar{x}\ \mathsf{else}\ M_{\mathrm{ff}} g \bar{x}\ \mathsf{fi}$$

est un terme qui définit f à partir de g.

Au contraire, supposons maintenant que $V = V_{H_f}$ est cohérent. Soit ϕ_i (pour $1 \leq i \leq m$) la $i^{\text{ème}}$ tranche de α_V. On définit $\hat{\phi}_i$ par

$$\hat{\phi}_i = \begin{cases} \phi_i & \text{si } \#(\mathsf{tr}(\phi_i)) < \#(\mathsf{tr}(f)) \\ \lambda \bar{x}.v & \text{pour } v \in \pi_2(\mathsf{tr}(\phi_i)) \text{ sinon} \end{cases}$$

D'abord, vérifions que $\hat{\phi}_i$ est bien définie, c'est-à-dire que si $\#(\mathsf{tr}(\phi_i)) \geq \#(\mathsf{tr}(f))$, alors $\pi_2(\mathsf{tr}(\phi_i))$ est un singleton. Si $\#(\mathsf{tr}(\phi_i)) \geq \#(\mathsf{tr}(f))$, alors $\#(\mathsf{tr}(\phi_i)) = \#(\mathsf{tr}(f))$, ce qui nous donne $\pi_1 \mathsf{tr}(\phi_i) = \pi_1 \mathsf{tr}(f)$. Comme V est un hyperarc de H_f c'est aussi un hyperarc de H_{ϕ_i} qui doit, puisque ϕ_i est sous-séquentielle, être monochrome.

Prouvons maintenant que les $\hat{\phi}_i$ sont g-définissables. Comme $\lambda\bar{x}.v$ est définissable, il ne reste à vérifier que la cas $\hat{\phi}_i = \phi_i$. On définit $\overline{\phi_i}$ par sa trace

$$\mathsf{tr}(\overline{\phi_i}) = \{v \in \mathsf{tr}(f) \mid \pi_1(v) \in \pi_1(\mathsf{tr}(\phi_i))\}$$

$\overline{\phi_i}(\bar{x})$ est égal à $f(\bar{x})$ quand $\phi_i(\bar{x}) \neq \bot$, sinon $\overline{\phi_i}(\bar{x}) = \bot$. Par le lemme 2.3.10, comme les ϕ_i sont sous-séquentielles, $\phi_i \preceq_{\text{def}} \overline{\phi_i}$. $\#(\mathsf{tr}(\overline{\phi_i})) = \#(\mathsf{tr}(\phi_i)) < \#(\mathsf{tr}(f))$ et, par le lemme 2.3.7, $\mathcal{TM}[H_{\overline{\phi_i}}, H_g] \neq \emptyset$. Par hypothèse d'induction, $\overline{\phi_i} \preceq_{\text{def}} g$. La transitivité de $\preceq_{\text{def}}$ nous donne $\hat{\phi}_i \preceq_{\text{def}} g$: soit M_i un terme qui définit $\hat{\phi}_i$ à partir de g.

Prouvons maintenant que nous avons défini un test de convergence de f à partir de g : pour tout $\bar{x} \in \mathsf{B}^l$,

$$f\bar{x}) \neq \bot \iff g([\![M_1]\!]g\bar{x}, \ldots, [\![M_m]\!]g\bar{x}) \neq \bot$$

La direction $\Longrightarrow$ est évidente, car les $\hat{\phi}_i$ sont des majorants des ϕ_i : s'il existe $\bar{v} \in \pi_1(\mathsf{tr}(f))$ tel que $\bar{v} \sqsubseteq \bar{x}$, alors $g([\![M_1]\!]g\bar{x}, \ldots, [\![M_m]\!]g\bar{x}) \sqsupseteq \alpha_V(\bar{v})$

Pour la direction opposée, supposons que pour tout $v \in \pi_1(\mathsf{tr}(f))$, $v \not\sqsubseteq x$. Par définition des $\hat{\phi}_i$, Par définition des $\hat{\phi}_i$, nous savons que pour tout $\bar{w} \in \alpha_V(V_{H_f})$, $([\![M_1]\!]g\bar{x}, \ldots, [\![M_m]\!]g\bar{x}) \leq \bar{w}$, puisque, sous l'hypothèse que $f(\bar{x}) = \bot$, nous avons que pour tout $1 \leq j \leq m$, si $[\![M_j]\!]g\bar{x} = b > \bot$, alors $\hat{\phi}_j = \lambda\bar{x}.\ b$, et donc que pour tout $\bar{w} \in \alpha(V_{H_f})$, $w_j = b$. Comme $V = V_{H_f}$ est un hyperarc, nous savons que $\#(\alpha_V(V)) \geq 2$, et, par minimalité des éléments de $\pi_1(\mathsf{tr}(g))$, on conclut que pour tout $w \in \pi_1(\mathsf{tr}(g))$ $([\![M_1]\!]g\bar{x}, \ldots, [\![M_m]\!]g\bar{x}) \not\sqsupseteq w$, et donc que $g([\![M_1]\!]g\bar{x}, \ldots, [\![M_m]\!]g\bar{x}) = \bot$.

Nous pouvons maintenant conclure la preuve, encore une fois selon la structure de H_f. Si V_{H_f} est monochrome – supposons par exemple que tous les sommets sont blancs – il est facile de vérifier que f est définie à partir de g grâce au terme :

$$\lambda g.\lambda\bar{x}.\mathsf{if}\ g(M_1 g\bar{x}) \ldots (M_m g\bar{x})\ \mathsf{then}\ \mathtt{tt}\ \mathsf{else}\ \mathtt{tt}\ \mathsf{fi}$$

Si V_{H_f} n'est pas monochrome, alors, il est facile de vérifier que

$$\forall x, y \in V_{H_f}\ C(x) = C(y) \Leftrightarrow C(\alpha_V(x)) = C(\alpha_V(y))$$

i.e. α_V agit comme l'identité ou la négation sur les couleurs : si on note ϵ le terme qui définit l'opération correspondante sur les booléens, on peut vérifier que

$$M = \lambda g\ \lambda\bar{x}.\ \epsilon(g(M_1 g\bar{x}) \ldots (M_m g\bar{x}))$$

■

2.3.5 Complétude pour les fonctions sous-séquentielles

La réciproque n'est en général pas vraie. Par exemple, on définit f, g par leur traces

$$\begin{cases} f\ \mathrm{tt}\ \mathrm{tt}\ \mathrm{tt}\ \bot\ \bot\ \bot = \mathrm{tt} \\ f\ \mathrm{ff}\ \bot\ \bot\ \mathrm{tt}\ \mathrm{tt}\ \bot = \mathrm{tt} \\ f\ \bot\ \mathrm{ff}\ \bot\ \mathrm{ff}\ \bot\ \mathrm{tt} = \mathrm{tt} \\ f\ \bot\ \bot\ \mathrm{ff}\ \bot\ \mathrm{ff}\ \mathrm{ff} = \mathrm{tt} \end{cases}$$

$$\begin{cases} g\ \bot\ \mathrm{tt}\ \mathrm{ff} = \mathrm{tt} \\ g\ \mathrm{ff}\ \bot\ \mathrm{tt} = \mathrm{tt} \\ g\ \mathrm{tt}\ \mathrm{ff}\ \bot = \mathrm{ff} \end{cases}$$

Tous les ensembles avec au moins trois points de H_f sont des hyperarcs, et H_g a un unique hyperarc, qui a trois points lui aussi, donc il n'y a pas de h-morphisme non trivial de H_d dans $\overline{H_g}$: au moins un hyperarc de H_f est envoyé sur l'hyperarc de H_g, et donc, quel que soit le choix pour le quatrième point de H_f, un hyperarc de H_f sera envoyé sur un doublet de points de H_g, qui n'est pas un hyperarc. Pourtant, g est le plus grand degré de définissabilité des fonctions stables (voir [Cur93] p. 334), et f est stable.

Pour les fonctions sous-séquentielles par contre, les hypergraphes et les morphismes temporisés sont complets. On énonce d'abord un lemme facile, qui lie les notions de relation de séquentialité et de cohérence linéaire :

Lemme 2.3.13

Soit $X = \{\bar{x}_1, \ldots, \bar{x}_n\} \subseteq \mathsf{B}^m$ et I un sous-ensemble de $\{1 \ldots n\}$. $\{\bar{x}_i\}_{i \in I}$ est linéairement cohérent si $(x_{ij}) \in \mathcal{S}^n_{B,B}$. De plus, X est cohérent si et seulement si pour chaque $1 \leq j \leq m$

$$\left(x_{j1}, \ldots, x_{jn}, \prod_{1 \leq 1 \leq n} x_{ji} \right) \in \mathcal{S}^{n+1}_{n,n+1}$$

ou $\mathcal{S}^{n+1}_{n,n+1}$ est la relation de Sieber $\mathcal{S}^{\{1 \ldots n+1\}}_{\{1 \ldots n\},\{1 \ldots n+1\}}$.

Théorème 2.3.14

Soient $f : \mathsf{B}^n \to \mathsf{B}$ et $g : \mathsf{B}^m \to \mathsf{B}$ deux fonctions sous-séquentielles telles que $\mathcal{TM}[H_f, H_g] = \emptyset$. Alors $f \not\trianglelefteq_{\mathrm{def}} g$.

Démonstration : La première remarque est qu'il existe un ensemble $A \in A_{H_f}$ tel qu'il n'y a pas de h-morphisme non trivial de $H_f{\restriction}_A$ dans $\overline{H_g}$. Pour la suite de la preuve, on choisit un tel A et on notera ses éléments $\bar{v}_1 \ldots \bar{v}_k$. Soient $A_1 \ldots A_l$ les arcs de $H_f{\restriction}_A$ et, pour $1 \leq j \leq l$, X_j l'ensemble des indices des éléments de A_j : $A_j = \{\bar{v}_k \mid k \in X_j\}$.

On considère la relation de séquentialité $k+1$-aire

$$S_A = \left(\bigcap_{1 \leq j \leq l} \mathcal{S}^{k+1}_{X_j, X_j} \right) \cap \mathcal{S}^{k+1}_{k,k+1}$$

On va prouver que g est invariante par S, et que f ne l'est pas : par le lemme fondamental des relations logiques $f \not\preceq_{\text{def}} g$.

Soit $V = (\bar{v}_1, \ldots, \bar{v}_k, \bigsqcup_{1 \leq j \leq k} v_j)$: par le lemme 2.3.13, pour $1 \leq j \leq l$, $V \in \mathcal{S}^{k+1}_{X_j,X_j}$ et $V \in \mathcal{S}^{k+1}_{k,k+1}$, donc $V \in S_A$. Par contre

$$\left(f(\bar{v}_1), \ldots, f(\bar{v}_k), f(\bigsqcup_{1 \leq j \leq k} \bar{v}_j) \right) \notin \mathcal{S}^{k+1}_{k,k+1}$$

car les k premiers éléments sont définis (les v_j sont dans la trace de f), alors que le dernier est $\perp$ ($\bigsqcup_{1 \leq j \leq k} \bar{v}_j$ ne peut pas majorer un v_j). On a donc prouvé que f n'était pas invariant par S_A.

Pour prouver que g est invariant par S_A, on raisonne par l'absurde : supposons qu'il existe $(\bar{w}_1, \ldots, \bar{w}_{k+1}) \in S_A$ telle que

$$(g(\bar{w}_1), \ldots, g(\bar{w}_{k+1})) \notin S_A$$

D'abord, $W \in \mathcal{S}^{k+1}_{X_j,X_j}$, donc $\{w_i \mid i \in X_j\}$ est cohérent. Comme g est sous-séquentielle, $\{g(w_i) \mid i \in X_j\}$ est également cohérent (proposition 2.3.9), et donc $(g(\bar{w}_1), \ldots, g(\bar{w}_{k+1})) \notin \mathcal{S}^{k+1}_{X_j,X_j}$ (lemme 2.3.13). Il faut donc que $g(W) \notin \mathcal{S}^{k+1}_{k,k+1}$, c'est-à-dire que

$$\begin{cases} \forall 1 \leq j \leq k, g(\bar{w}_j) \neq \perp \\ \exists j, j' \leq k+1, g(\bar{w}_j) = g(\bar{w}_{j'}) \end{cases}$$

Comme g est sous-séquentielle et $\{\bar{w}_1, \ldots, \bar{w}_k\}$ est cohérent (lemme 2.3.13) $\forall j, j' \leq k, g(\bar{w}_j) = g(\bar{w}_{j'})$. Il existe donc un $b \in \mathsf{B}$ tel que

$$\begin{cases} \forall 1 \leq j \leq k, g(\bar{w}_j) = b \\ g(\bar{w}_{k+1} \neq b \end{cases}$$

Il existe donc pour $1 \leq i \leq k$ un élément $z_i \in \pi_1(\mathsf{tr}(g))$ tel que $z_i \sqsubseteq \bar{w}_i$. En résumé :

- l'ensemble $\{z_1, \ldots, z_k\}$ n'est pas un singleton, sinon $g(\bar{w}_{k+1}) = b$ (par définition de $\mathcal{S}^{k+1}_{k,k+1}$, $\bar{w}_{k+1} \sqsupseteq \bigsqcup_{1 \leq j \leq k} \bar{w}_j$),
- pour tout $1 \leq j \leq l$, l'ensemble $\{z_i \mid i \in X_j\}$ est cohérent (c'est un minorant pour l'ordre d'Egli-Milner de l'ensemble cohérent $\{w_i \mid i \in X_j\}$)
- C_{H_f} est constante sur A (f est sous-séquentielle).

La fonction $\alpha : A \to H_g$ définie par $\alpha_A(\bar{v}_i) = z_i$ est donc un h-morphisme non trivial de $H_f{\restriction}_A$ dans H_g, ce qui contredit l'hypothèse. ■

2.3.6 Complexité

On appelle hypergraphes saturés les hypergraphes dont l'ensemble des hyperarcs est fermé par union. On va réduire le problème "Set-splitting" à la recherche d'un h-morphisme entre deux hypergraphes saturés, puis ramener celui-ci à la recherche d'un morphisme temporisé entre deux hypergraphes saturés. On aura alors prouvé que décider de l'existence des h-morphismes et des morphismes temporisés est un problème NP-dur.

D'un autre côté, décider de l'existence des h-morphismes et décider des morphismes temporisés sont des problèmes NP, puisque vérifier qu'une fonction préserve les hyperarcs et deviner une fonction sont des procédures polynomiales en la taille des hypergraphes.

Finalement, on aura bien montré que décider de l'existence des h-morphismes et des morphismes temporisés sont des problèmes NP-complets.

Commençons par définir le problème Set Splitting (référence [SP4] 3 dans [GJ79]) :

Instance : un ensemble fini U et des sous-ensembles $A_1, ..., A_k$ de de U.

Question : existe-t-il une partition $U = U_1 \cup U_2$ telle que pour tout j, $A_j \cap U_1 \neq \emptyset$ et $A_j \cap U_2 \neq \emptyset$?

Étant donnée une instance $U, A_1 \dots A_k$ de Set Splitting, on suppose que pour tout i, j, $A_i \not\subseteq A_j$: dans le cas contraire, on retire A_j, ce qui n'affecte pas l'existence d'une solution au problème. On définit alors l'hypergraphe saturé H, dont les sommets sont les éléments de U et les arcs minimaux les A_i.

On notera $\mathbf{2}$ l'hypergraphe avec deux sommets $0, 1$ et un arc unique, l'ensemble des sommets :

Lemme 2.3.15

L'instance de Set Splitting a une solution si et seulement s'il existe un h-morphisme de H dans $\mathbf{2}$.

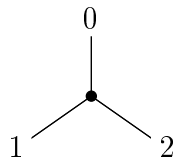
Démonstration : Si on a un h-morphisme $f : H \to \mathbf{2}$, on définit $U_1 = f^{-1}(0)$ et $U_2 = f^{-1}(1)$. Par construction, $U_1 \cup U_2 = U$, et pour tout i, $f(A_i) = \{0, 1\}$ car A_i est un hyperarc de H, donc $U_1 \cap A_i \neq \emptyset$ et $U_2 \cap A_i \neq \emptyset$.

Réciproquement, si U_1, U_2 est une solution, on définit $f : x \mapsto j$ t.q. $x \in U_j$. f est un h-morphisme de H dans $\mathbf{2}$: f est bien définie car $U_1 \cap U_2 = \emptyset$,

et si A est un hyperarc de H, alors $A = A_i$ pour un certain i et, comme $A_i \cap U_j \neq \emptyset$ $(j = 1, 2)$, $f(A) = \{0, 1\}$ qui est un hyperarc **2**. ■

Il reste à réduire l'existence d'un h-morphisme entre un hypergraphe saturé et **2** à l'existence d'un morphisme temporisé.

Étant donné un hypergraphe saturé H et $* \notin V_H$, soit H^* l'hypergraphe (saturé) défini par $V_{H^*} = V_H \cup \{*\}$ et $A_{H^*} = \{A \cup \{*\} \mid A \in A_H\}$ Soit **3** l'hypergraphe avec trois sommets $0, 1, 2$ et un arc unique, l'ensemble des sommets :



Lemme 2.3.16

Soit H un hypergraphe saturé. Il existe un h-morphisme de H vers **2** si et seulement s'il existe un morphisme temporisé de H^* vers **3**.

Démonstration : Soit H un hypergraphe saturé. D'abord, si f est un h-morphisme de H vers **2**, on définit, pour $B \in A_{H^*}$,

$$\alpha_B(x) = \begin{cases} f(x) & \text{si } x \neq * \\ 2 & \text{si } x = * \end{cases}$$

$\{\alpha_B\}$ est évidemment un morphisme temporisé de H^* vers **3**.

Pour prouver la réciproque, on se donne un morphisme temporisé $\{\alpha_B\}$ de H^* vers **3**, et on définit un h-morphisme de H dans **2** qui "découpe" chaque arc de H.

L'idée est la suivante : étant donnés deux arcs $A \subset B$ de H^*, on sait que soit $\alpha_B(A) = \{\alpha_B(*)\}$ soit $\alpha_B(A) = \{0, 1, 2\}$. Dans le dernier cas, on construit facilement un morphisme (partiel) de H vers **2** qui découpe $A \setminus \{*\}$, et dans le premier cas, il doit y avoir un arc $C \subset \alpha_B^{-1}\{*\}$ tel que $\alpha_C(A) = \{0, 1, 2\}$. Dans les deux cas, tous les arcs seront finalement découpés : en recollant les morphismes partiels, on obtient un h-morphisme complet.

Formellement, on définit une suite décroissante d'arcs de H^* par :

- $A_0 = \bigcup\{A \in A_{H^*}\}$
- $A_{n+1} = \bigcup\{A \in A_{H^*} \mid A \subseteq \alpha_{A_n}^{-1}\{*\}\}$

Pour tout n tel que $A_n \neq \emptyset$, $A_{n+1} \subset A_n$ car α_{A_n} n'est pas trivial. Soit l_0 le plus petit indice tel que $A_{l_0} = \emptyset$ (il faut noter que $l_0 \leq |A_{H^*}|$, car au moins un arc est est coupé à chaque étape).

Soit

$$\begin{aligned} i_n &= \alpha_{A_n}(*) \\ j_n &= (i_n + 1) \ mod \ 3 \\ k_n &= (i_n + 2) \ mod \ 3 \end{aligned}$$

On définit deux ensembles disjoints de V_H :

$$U_1 = \bigcup_{s=0}^{l_0-1} \alpha_{A_s}^{-1}\{j_s\}, \qquad U_2 = \bigcup_{s=0}^{l_0-1} \alpha_{A_s}^{-1}\{k_s\}$$

Il n'est pas difficile de vérifier que U_1, U_2 séparent tous les arcs de H : on définit

$$f(x) = \begin{cases} 0 & \text{si } x \in U_1 \\ 1 & \text{sinon} \end{cases}$$

et on vérifie que pour tout $A \in A_H$, $f(A) = \{0, 1\}$.

D'abord, il faut noter que pour un certain n, $A \subseteq A_{n-1}$ et $A \not\subseteq A_n$, car A_n est une suite décroissante. De plus, $0 < n < l_0$, car A_0 est l'union de tous les arcs de H^*, et $A_{l_0} = \emptyset$.

Cela signifie que $\alpha_{A_{n-1}}(A \cup \{*\}) = \{0, 1, 2\}$, et donc que $A \cap U_1 \neq \emptyset$ et $A \cap U_2 \neq \emptyset$. On obtient, $f(A) = \{0, 1\}$, ce qui conclut la preuve. ■

2.3.7 Conclusions

Ce travail complète les résultats antérieurs de Bucciarelli et Malacaria [Buc97, BM02] : la caractérisation géométrique de la définissabilité relative au premier ordre s'avère complète pour les fonctions sous-séquentielles, qui représentent une fraction importante des fonctions non définissables du modèle de Scott de PCF finitaire.

Le résultat sur la complexité –décider de la définissabilité relative est NP-complet– est nouveau : Stoughton [Sto94] avait proposé un algorithme, mais n'avait pas étudié sa complexité.

Il reste que cette technique reste limitée : en particulier, elle ne s'applique pas aux ordres supérieurs, et elle est incomplète pour les fonctions qui ne sont pas sous-séquentielles. Il faudrait sans doute, pour approfondir ces résultats, des représentations plus complexes des fonctions, et des notions de morphismes plus élaborées.

2.4 Définissabilité dans les modèles stables

Les résultats précédents concernaient la définissabilité dans les modèles de Scott. Dans les modèles stables, la non-définissabilité n'est plus due à des

comportements "parallèles", mais à un examen trop précis de la façon dont la fonction calcule un résultat : ainsi, dans notre exemple qui teste si son argument est strict, la fonction observe d'un peu trop près le déroulement du calcul. On peut donc parler, pour les degrés de définissabilité dans les modèles stables, de degrés d'intensionalité, au contraire des degrés de parallélisme des modèles extensionnels.

Paolini [Pao04] a montré que les éléments du modèle stable étaient tous définissables à partir des fonctions suivantes :

- $\mathsf{test} : (N_\perp \to N_\perp) \to N_\perp$, notre exemple,
- $\mathsf{cond}_3 : N_\perp^6 \to N_\perp$, une version à six arguments de la fonction de Gustave.

En ce sens le modèle stable n'est pas exempt de comportements "parallèles" : la fonction de Gustave est essentiellement une version de por avec plus d'arguments. On va donc, pour mieux comprendre les problèmes de séquentialité, se tourner vers le modèle fortement stable, qui généralise les définitions du modèle stable pour ne conserver que les fonctions séquentielles.

Dans [Lon02], Longley expose un modèle de PCF construit par réalisabilité, isomorphe au modèle fortement stable. Les techniques qu'il utilise lui permettent de construire une fonctionelle H, et de prouver qu'elle est universelle, c'est-à-dire qu'elle permet de définir tous les autres éléments du modèle. Sa construction est assez technique, et ce que fait H ne nous a pas semblé très intuitif. Dans cette section, on essaiera de présenter un programme qui implémente H, avec comme objectif de mieux comprendre les possibilités d'observation des fonctionnelles fortement stables.

Le problème n'est en effet pas simple : les possibilités ajoutées par le modèle n'ont rien à voir avec le parallélisme (le but même de la stabilité forte est bien de caractériser les fonctions séquentielles), mais plutôt avec l'examen de la façon de calculer des programmes : ainsi, on pourra savoir si une fonction utilise son argument, puisque la fonction qui distingue (fortement) une fonction stricte d'une fonction constante est dans le modèle. Pourtant, et c'est la différence avec, par exemple, les algorithmes séquentiels ou le modèle de jeux sans parenthésage, le modèle reste extensionnel : il est impossible de distinguer des programmes qui font les mêmes choses dans des ordres différents (cf. goûteur de "et"). C'est précisément le coeur de la fonction H : elle permet de "décompiler" une fonction, sous réserve que l'utilisation du programme extrait ne dépende que du comportement extensionnel de la fonction.

Cette section ne contient pas de nouveaux résultats, et encore moins de preuves : il s'agit seulement d'éclairer la définition de H, en utilisant une syntaxe concrète. Toutes les aspects formels sont dans le rapport de Longley sur les fonctionnelles séquentiellement réalisables [Lon02]. Il faut également noter que Longley étudie une fonctionnelle proche du quote décrit ici dans

[Lon03].

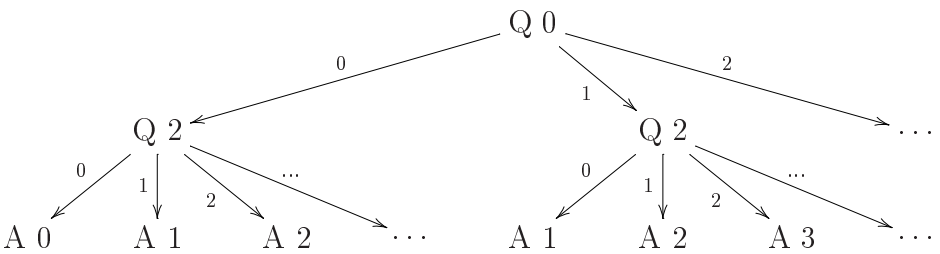
2.4.1 Arbres de décision

Le modèle de Longley est construit par réalisabilité sur un ensemble d'arbres de décisions. Au type $\bar{2} = (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$, ces arbres représentent les échanges de la fonction avec son environnement : elle peut répondre un entier n (noté $!n$) ou demander la valeur de son argument appliqué à un entier m (Longley le note $?m$).

On appelle node le type correspondant en ML :

```
type node =
    Q of int
  | A of int
```

Par exemple, la fonction fun g->g(0)+g(2) est représentée par l'arbre[1] :



Comme cet arbre est infini, on va le programmer comme une fonction des chemins (les réponses aux questions déja posées) dans les questions/réponses :

```
let tree_f = function
  []    -> Q 0
| [n]   -> Q 2
| [n;m] -> A (n+m)
| _     -> diverge()
```

L'intuition veut que les réponses soient des feuilles dans l'arbre : quand la fonction a répondu, on ne peut pas continuer. On ne considère donc que les arbres bien formés :

Définition 2.4.1

> Un arbre de décision est une fonction T des listes d'entiers dans les actions (question ou réponse), telle que si $T\,[x_1, \ldots, x_n]$ est défini, alors pour tous les $k < n$, $T[x_1, \ldots, x_k]$ est défini et c'est une question.
>
> Un arbre de décision n'est pas redondant si, quand $T[x_1, \ldots, x_n]$ est

[1] Si + évalue d'abord à gauche

défini, tous les noeuds rencontrés sont des questions différentes.

À partir d'un arbre de décision, on peut "simuler" la fonction correspondante : il suffit d'exécuter les coups demandés :

- on commence avec une liste vide $p = []$
- si $T(p) = Qn$ et $h(n) = m$, on ajoute m à la fin de p et on recommence,
- si $T(p) = An$, la valeur finale est n.

Cet algorithme peut être écrit en PCF, il existe donc dans le modèle. Appelons le eval. Le programme Caml correspondant a le type tree $\to (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$:

```
let eval t h =
  let rec path l = match t l with
      Q n -> path (l@[h n])
    | A n -> n
  in
  path []
```

Jusqu'ici, on a seulement défini un interpréteur pour le langage des arbres de décision. La partie intéressante du travail est de construire ces arbres.

2.4.2 quote

On voudrait un algorithme quote : $((\mathsf{N} \to \mathsf{N}) \to \mathsf{N}) \to$ tree qui produise un programme à partir d'une fonction, tel que eval$\circ$quote $= id$. L'intuition est qu'on suit pas à pas le déroulement du programme et qu'on construit l'arbre en conséquence. quote n'est pas extensionnel, puisque l'arbre de décision va refléter l'ordre des questions. On ne peut donc pas l'écrire en PCF. En Caml, par contre, l'utilisation combinée d'exceptions et de références permet de suivre pas à pas le programme : on lui donne une "fausse" fonction comme argument, qui fournit les réponses prévues (le chemin de l'arbre) tant qu'il y en a, et qui lève une exception quand on arrive au bout du chemin, pour récupérer la dernière question posée :

```
exception OneMoreQuestion of int

let quote f path =
  let remaining_path = ref path in
  let countdown n =
    match !remaining_path with
        [] -> raise (OneMoreQuestion n)
      | a::l -> remaining_path := l; a
```

```
in
try
  let val = f countdown in
  if !remaining_path=[]
  then A val
  else diverge()
with OneMoreQuestion n -> Q n
```

quote prend deux arguments, une fonction f :$(\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$ et une liste d'entiers path (le chemin) et retourne une action (le nœud correspondant à ce chemin). On met la liste dans une fonction comme expliqué précédemment, pour en faire une fonction countdown :$\mathsf{N} \to \mathsf{N}$. À chaque appel, elle renvoie la tête de liste restante, c'est-à-dire la réponse correspondant à cette question dans le chemin. Il faut noter que l'argument x n'est utilisé que lorsque la liste est vide, i.e. quand on est arrivé au bout du chemin demandé : on interrompt alors le calcul et on retourne la question posée x.

Le corps de la fonction applique f à countdown. Si le calcul termine normalement, c'est que f a renvoyé une valeur : on vérifie qu'on est bien au bout du chemin, et on renvoie cette valeur, mais présentée comme une action. Si on n'était pas au bout du chemin, alors ce chemin n'existe pas dans l'arbre, et on ne renvoie rien. Si une exception est levée pendant l'exécution de f, c'est qu'elle a posé une nouvelle question qu'on retourne, encore une fois présentée comme une action.

L'arbre produit pour fun g->g(0)+g(0) pose deux fois la question Q 0, il est donc différent de l'arbre produit par fun g->2*g(0), alors que ces fonctions ont la même dénotation. quote ne peut donc pas faire partie du modèle. Il nous faut en fait une version qui donne des arbres non-redondants : on augmente la fonction countdown pour qu'elle réponde automatiquement aux questions posées précédemment.

```
exception QuoteStop of int

let irredundant_quote f path =
  let remaining_path = ref l
  and already_asked = ref []
  in
  let countdown already_asked n =
    try
      assoc x !already_asked
    with Not_found ->
      match !remaining_path with
        []   -> raise (QuoteStop n)
```

```
    | a::l -> remaining_path := l;
            already_asked := (n,a)::(!already_asked);
            a
in
try
  let m=f countdown in
  if !remaining_path=[]
  then A m
  else loop()
with QuoteStop n -> Q n
```

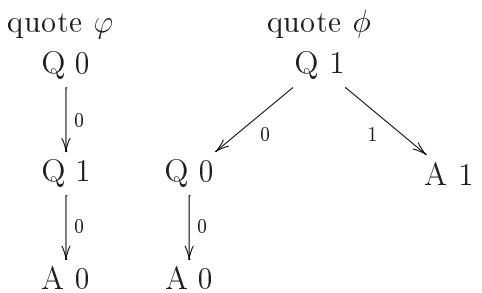
Pour notre exemple :

$$\text{quote f l} = \begin{cases} \text{Q } 0 & \text{if l=[]} \\ \text{Q } 2 & \text{if l=[}x\text{]} \\ \text{A}(x+y) & \text{if l=[}x\text{;}y\text{]} \\ \text{rien} & \text{dans les autres cas} \end{cases}$$

Si f=fun g->g(2)+g(0), les questions sont échangées :

$$\text{quote f l} = \begin{cases} \text{Q } 2 & \text{if l=[]} \\ \text{Q } 0 & \text{if l=[}x\text{]} \\ \text{A}(x+y) & \text{if l=[}x\text{;}y\text{]} \\ \text{rien} & \text{dans les autres cas} \end{cases}$$

Ces exemples montrent que quote n'est pas extensionnel, et donc ne fait pas partie du modèle fortement stable. On pourrait espérer que mettre un ordre sur les questions et imposer qu'on ne pose qu'une fois les questions résoudrait ce problème, mais même avec ces conditions, quote ne serait pas croissante pour l'ordre stable. En effet, supposons que quote ' est une nouvelle version de quote qui produit un arbre dans lequel les questions sont posées au plus une fois et dans l'ordre croissant. On définit des fonctions $f, g : \mathsf{N} \to \mathsf{N}$ par leur traces : $\mathsf{tr}(f) = \{(0,0),(1,0)\}$ et $\mathsf{tr}(g) = \{(1,1)\}$. Pour distinguer f et g, il faut tester leurs valeurs en 1. Soient $\varphi, \phi : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$ des fonctions définies par $\mathsf{tr}(\varphi) = \{(f,0)\}$ et $\mathsf{tr}(\phi) = \{(f,0),(g,1)\}$. En appliquant quote ', on obtient :



φ et ϕ sont fortement stables, et $\varphi < \phi$. Pourtant, quote φ [] = Q 0 et quote ϕ [] = Q 1, ce qui prouve que quote n'est pas croissante pour l'ordre stable, et donc ne peut pas faire partie du modèle. Il n'y a donc aucun espoir de trouver une telle fonction, même dans ses versions finitaires, dans le modèle fortement stable : les observations faites avec quote sont trop fines.

2.4.3 La fonctionnelle H

On va donc prendre le problème à l'envers : au lieu de chercher une fonction intermédiaire (plus faible que quote), on va donner quote en argument aux fonctions qui voudraient utiliser des informations intensionnelles, en vérifiant qu'ils ne se servent pas d'observations "interdites". L'idée est la suivante : étant donnée une fonction des arbres dans les entiers, on veut en faire une fonctionnelle des fonctions de type $(\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$ dans les entiers. On voudrait écrire $\lambda\phi.\lambda f.\phi(\text{quote}f)$, mais en vérifiant que ϕ renverrait la même valeur sur tous les arbres qui implémentent la fonction f.

Le plan est donc :

1. prendre une fonction f : tree $\to \mathsf{N}$,
2. prendre une fonctionnelle $g : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$,
3. trouver tous les arbres qui implémentent g,
4. vérifier que f à la même valeur en chacun de ces arbres,
5. si oui, renvoyer cet entier, sinon diverger.

Deux problèmes se posent : comment trouver tous les arbres qui implémentent g ? Que faire s'il y en a une infinité ?

La première question est assez simple à résoudre : deux arbres implémentent la même fonction s'ils posent les même questions, éventuellement en changeant leur ordre. Il suffit donc d'obtenir un arbre grâce à quote , et de calculer ses permutations. Bien sûr, comme cet arbre est en général infini, on ne peut même pas le calculer entièrement ! En fait, f n'utilise qu'une partie finie de l'arbre pour terminer (elle est dans la trace de f). Il suffit donc de la trouver, et de calculer ses permutations (qui sont en nombre fini). Le plan est donc plus précisément :

1. prendre une fonction f : tree $\to \mathsf{N}$,
2. prendre une fonctionnelle $g : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$,
3. trouver un arbre t qui implémente g en utilisant quote ,
4. appliquer f à t, et récupérer une valeur n,
5. trouver la partie finie t_0 de t utilisée par f,
6. calculer les permutations t_i de t_0,

7. vérifier que $ft_i = n$ pour tout les t_i,
8. si oui, renvoyer n, sinon diverger.

Pour calculer t_0, on va utiliser la fonction "modulo" de type $((\mathsf{N}\ \mathrm{list} \to \mathsf{N}) \to \mathsf{N}) \to (\mathsf{N}\ \mathrm{list} \to \mathsf{N}) \to \mathsf{N}\ \mathrm{list\ list}$, qui renvoie la liste des arguments (listes d'entiers) que la première fonction donne à la deuxième :

```
let modulo f t =
  let l = ref [] in
  let t' x = l := x::(!l); t x in
    f t';
    list_to_set !l
```

Pour rester extensionnel, il a fallu transformer la liste des arguments donnés à la fonction en ensemble non ordonné, pour éliminer les répétitions et les considérations d'ordre : list_to_set est une fonction qui renvoie la liste triée sans doublons. À partir de cette liste de chemins, il est facile de reconstruire le sous arbre fini t_O utilisé par f. Appelons tree_modulo cette fonction.

Il reste à écrire la fonction qui calcule les permutations de cet arbre t_0 : il s'agit en fait, à chaque nœud de l'arbre, de chercher les questions posées dans toutes les branches. Chacune de ces questions peut se trouver à la racine du sous arbre considéré. Cette fonction est assez longue, on ne l'écrira pas ici. Appelons-la simplement permut_tree.

Nous avons maintenant tous les éléments pour définir la fonction H :

```
let h f g =
  let t = quote g in
  let t0 = tree_modulo f t0 in
  let trees = permut_tree t0 in
  let results = map f trees in
    match results with
        x::l ->
          if List.for_all (fun y->x=y) l
          then x
          else loop()
```

2.4.4 Conclusions

Cette étude montre, si la preuve devait encore en être faite (Royer [Roy04] a montré que toutes les implémentations de H étaient exponentielles) que le pouvoir expressif des fonctions séquentiellement réalisables (ou fortement stables) est assez difficile à saisir. Alors que dans le cas d'un langage plus riche

comme PCF+catch, les possibilités sont claires (le modèle des algorithmes séquentiels a la propriété de définissabilité [CF92]), imposer l'extensionalité rend les choses beaucoup plus complexes : finalement, on a le droit de suivre le programme pas à pas, tant qu'on ne se sert pas "vraiment" des informations obtenues de cette façon. La programmation reste "fonctionnelle" (puisqu'il n'y a pas d'effets de bords, et que le raisonnement équationnel reste valable), mais dans un sens assez particulier : Longley [Lon99] donne quelques exemples d'utilisation de ces nouvelles possibilités.

Chapitre 3

Allocation dynamique

Cette partie reprend le travail que nous avons effectué avec Nick Benton, chez Microsoft Research à Cambridge, pendant l'été 2004. Les techniques décrites dans l'introduction (sémantique opérationnelle, sémantique dénotationnelle, preuve d'adéquation, quotient par une relation logique etc.) sont ici appliquées à un langage plus ambitieux et plus réaliste : au lieu de PCF, nous avons modélisé un langage fonctionnel en appel par valeur avec des variables allouées dynamiquement. Ces nouvelles constructions rendent la frontière entre observations par les contextes possibles et impossibles beaucoup plus difficile à saisir : on verra que la notion de secret devient prépondérante.

La recherche de bons modèles et de bons principes de preuve pour les langages avec des variables modifiables a une longue histoire [OT97, TG00]. Pour les langages du premier ordre impératifs, avec des variables globales de types simples (entiers, booléens, n-uplets etc.), la sémantique dénotationnelle qui représente la mémoire comme une fonction des adresses dans les valeurs et la logique de Floyd-Hoare sont satisfaisantes. Quand le langage devient plus riche, par contre, la situation est plus complexe : il n'existe pas encore de représentation mathématique complètement satisfaisante des variables telles qu'on les rencontre dans les langages courants depuis le milieu des années 60.

Si on se restreint aux langages "sûrs" et séquentiels, on peut dégager trois axes :

- Le langage de base : le premier ordre avec des boucles est plus facile que l'ordre supérieur sans récursion, qui lui-même est plus facile que l'ordre supérieur avec récursion.
- La visibilité et la durée de vie des variables est aussi un facteur important de complexité : les variables globales sont faciles à représenter, mais les variables locales (comme dans Algol), et plus encore les variables allouées dynamiquement sur le tas, sont complexes.

– Les types des valeurs qui peuvent être placées en mémoire : les valeurs de type de base (comme les entiers) posent moins de problèmes que les références, qui à leur tour sont plus faciles que les types d'ordre supérieur (comme les fonctions ou les procédures).

Dans ce travail, nous avons visé les situations intermédiaires : nous avons exclu les types récursifs, et interdit l'enregistrement des fonctions, mais nous considérons un λ-calcul complet avec des variables allouées dynamiquement. Nous obtenons donc un langage assez proche de ML [MTHM97], mais où on ne peut pas stocker des fonctions en mémoire. Nous ne traitons pas le cas des objets, mais la combinaison de l'allocation dynamique et des fonctions d'ordre supérieur offre un encodage raisonnable de la programmation orientée objets, ce qui laisse espérer que les techniques développées ici pour la famille de ML s'appliquent aussi aux langages à objets.

La propriété de base d'une sémantique dénotationnelle recherchée est l'adéquation : l'égalité dans le modèle doit impliquer l'équivalence opérationnelle. Bien que l'adéquation complète soit un but évident, elle est extrêmement difficile à obtenir, en particulier pour ces langages. Ce qu'on recherche est plutôt un bon compromis entre la simplicité et la précision : on veut tout simplement pouvoir prouver certaines équivalences, ou valider certains raisonnements.

La théorie des domaines traditionnelle offre des modèles satisfaisants (comme on l'a vu) pour les λ-calculs typés, et donc, par transitivité, pour tous les langages dont on peut décrire le comportement par un interpréteur écrit en λ-calcul typé. Le problème est que les modèles pour les langages à état obtenus de cette façon sont très loin d'être complètement adéquats : même des équivalences triviales sont fausses dans le modèle. Ceci est en partie dû au fait qu'on a décrit l'implémentation de la mémoire et des primitives qui y sont associées.

Pour construire des modèles plus abstraits, on peut soit inclure les équivalences dans la construction du modèle, soit quotienter un modèle existant. Dans ce travail, nous avons à la fois ajouté une structure spécifique aux domaines et quotienté le modèle obtenu.

La section 3.1 introduit le langage décrit : c'est un langage monadique en appel par valeur d'ordre supérieur, avec des références sur des variables allouées dynamiquement, qui peuvent contenir des entiers ou d'autres références. Dans la section 3.2, on définit une sémantique dénotationnelle en utilisant une monade de continuations sur des FM-cpos (ordres partiels complets avec une action du groupe des permutations des noms). L'utilisation des FM-cpos est inspirée des travaux de Shinwell et Pitts sur FreshML [SP05]. Elle ressemble en pratique beaucoup à la théorie des domaines traditionnelle : bien qu'il soit en théorie équivalent d'utiliser des pullbacks qui préservent les foncteurs de la catégorie des ensembles finis et des injections dans Cpo, il est beaucoup plus

agréable de travailler avec des FM-cpos. Cette interprétation nous donne un modèle élégant de l'allocation dynamique, mais, comme on le montre dans la section 3.2.4, ne permet pas de prouver la plupart des équivalences intéressantes.

Dans la section 3.3, on définit des relations logiques sur notre modèle de base. Elle sont paramétrées par des relations entre les états, qui spécifient quelle partie de la mémoire est accessible et un invariant (disjoint) qui doit être préservé par le contexte. Cette relation est inspirée par les travaux de Pitts et Stark [PS98] et de Reddy et Yang [RY04]. Le premier applique une relation à une sémantique opérationnelle pour un langage où seuls les entiers peuvent être enregistrés, et le deuxième utilise une catégorie de foncteurs pour un langage où tous les programmes terminent, en imposant une séparation plus forte entre les deux parties de la mémoire (visible et cachée).

On prouve que tous les termes sont invariants par cette relation, et, dans la section 3.4, on utilise ce résultat pour prouver quelques équivalences complexes à propos de l'allocation dynamique. On montre aussi que notre relation est incomplète pour l'équivalence observationnelle.

3.1 Le langage

3.1.1 Types et syntaxe

Comme on utilise une version monadique du lambda-calcul, on distingue les types des valeurs, notés A, et des programmes, de la forme $\mathbf{T}(A)$. Les types enregistrables forment un sous ensemble des valeurs, et on les note σ.

$$\begin{aligned} A &::= \mathsf{U} \mid \mathsf{N} \mid \sigma \text{ ref} \mid A \to \mathbf{T}(A) \\ \sigma &::= \mathsf{N} \mid \sigma \text{ ref} \\ \gamma &::= A \mid \mathbf{T}(A) \end{aligned}$$

Les fonctions prennent des valeurs en arguments, et renvoient des programmes. On se donne un ensemble infini (dénombrable) $\mathbb{L}$ d'adresses, qui représentent les cellules mémoire. On utilisera ℓ pour désigner les adresses en mémoire. Les types pour les états de la mémoire sont des fonctions finies des adresses dans les types enregistrables.

$$\Delta \ ::= \ \ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n$$

Les termes du langage sont aussi divisés en valeurs (notées V), et en

programmes (notés P) :

$$\begin{array}{lll} V & ::= & x \mid \underline{n} \mid \underline{\ell} \mid () \mid \mathsf{rec}\ f\ (x : A) = P \\ P & ::= & (V)V \mid \mathsf{let}\ x = V\ \mathsf{in}\ P \mid \mathsf{val}\ V \mid \mathsf{ref}\ V \mid !V \mid V := V \\ & & \mid V = V' \mid \mathsf{succ}\ V \mid \mathsf{pred}\ V \mid \mathsf{if}\ V\ \mathsf{then}\ P\ \mathsf{else}\ P\ \mathsf{fi} \\ G & ::= & P \mid V \end{array}$$

En général, le programme d'origine ne contient pas d'adresses ℓ, mais comme elles apparaissent pendant la réduction, nous les avons incluses dans le langage. À part les constructions "let" et "val", inspirées des travaux de Moggi, la syntaxe est proche de celle de ML. L'opération = teste l'égalité des pointeurs. Elle est en fait définissable (il suffit de modifier une parmi deux variables, et de voir si l'autre a changé ou pas), mais nous avons choisi de l'inclure quand même, car elle ne le serait plus si le type U était enregistrable.

Les règles de typage sont présentées en figure 3.1.

On utilisera aussi les notations suivantes :

$$\begin{array}{lll} \lambda x : A.P & = & \mathsf{rec}\ f\ (x : A) = P \quad \text{pour } f \notin fv(M) \\ M; N & = & \mathsf{let}\ x = M\ \mathsf{in}\ N \qquad\quad \text{avec } x \notin fv(N) \\ \Omega & = & (\mathsf{rec}\ f\ (x : \mathsf{U}) = f\ x)\ () \end{array}$$

Les états de la mémoire sont des fonctions finies de l'ensemble des adresses $\mathbb{L}$ dans l'union disjointe des entiers et des adresses :

$$\Sigma \in \mathbb{L} \rightharpoonup_{fin} \mathbb{Z} + \mathbb{L}$$

On note $\mathsf{in}_{\mathbb{Z}}$,$\mathsf{in}_{\mathbb{L}}$ les injections correspondantes, et $\Sigma[\ell \mapsto \mathsf{in}_{\mathbb{Z}} n]$ (resp. $\mathsf{in}_{\mathbb{L}}\ell$) la modification de l'état Σ. On note Σ, Σ' l'union d'états s'ils ont des domaines disjoints. $locs(G)$ est l'ensemble des adresses qui apparaissent dans le terme G, et $locs(\Sigma)$ est l'ensemble des adresses connues par l'état Σ : $\mathbb{L}(\Sigma) = dom(\Sigma) \cup \{\ell \in \mathbb{L} \mid \exists \ell'.\Sigma(\ell') = in_{\mathbb{L}}\ell\}$.

Définition 3.1.1 (États typés et équivalence)

Si Σ et Σ' sont des états et si $\Delta = \ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n$ est un type d'état, on définit

$$\Sigma \sim \Sigma' : \Delta \iff \forall 1 \leq i \leq n.\ \Sigma \sim \Sigma' : (\ell_i : \sigma_i)$$

où

$$\begin{array}{rcl} \Sigma \sim \Sigma' : (\ell : \mathsf{N}) & = & \exists n \in \mathbb{Z}.\Sigma\ell = \mathsf{in}_{\mathbb{Z}} n = \Sigma'\ell \\ \Sigma \sim \Sigma' : (\ell : \sigma'\ \mathrm{ref}) & = & \exists \ell' \in \mathbb{L}.\Sigma\ell = \mathsf{in}_{\mathbb{L}}\ell' = \Sigma'\ell \ \wedge\ \Sigma \sim \Sigma' : (\ell' : \sigma'). \end{array}$$

On dit qu'un état Σ a le type Δ, ce qu'on note $\Sigma : \Delta$, si $\Sigma \sim \Sigma : \Delta$.

Quelques faits sur le typage :

$$[\text{ty-unit}]\ \frac{}{\Delta;\Gamma \vdash () : \mathsf{U}} \qquad [\text{ty-int}]\ \frac{}{\Delta;\Gamma \vdash \underline{n} : \mathsf{N}}$$

$$[\text{ty-var}]\ \frac{}{\Delta;\Gamma, x : A \vdash x : A} \qquad [\text{ty-loc}]\ \frac{}{\Delta, \ell : A;\Gamma \vdash \underline{\ell} : A\ \mathrm{ref}}$$

$$[\text{ty-rec}]\ \frac{\Delta;\Gamma, f : A \to \mathbf{T}(B), x : A \vdash P : \mathbf{T}(B)}{\Delta;\Gamma \vdash \mathsf{rec}\ f\ x = P : A \to \mathbf{T}(B)}$$

$$[\text{ty-val}]\ \frac{\Delta;\Gamma \vdash V : A}{\Delta;\Gamma \vdash \mathsf{val}\ V : \mathbf{T}(A)}$$

$$[\text{ty-let}]\ \frac{\Delta;\Gamma \vdash P : \mathbf{T}(A) \quad \Delta;\Gamma, x : A \vdash P' : \mathbf{T}(B)}{\Delta;\Gamma \vdash \mathsf{let}\ x = P\ \mathsf{in}\ P' : \mathbf{T}(B)}$$

$$[\text{ty-app}]\ \frac{\Delta;\Gamma \vdash V : A \to \mathbf{T}(B) \quad \Delta;\Gamma \vdash V' : B}{\Delta;\Gamma \vdash (V)V' : \mathbf{T}(B)}$$

$$[\text{ty-if}]\ \frac{\Delta;\Gamma \vdash V : \mathsf{N} \quad \Delta;\Gamma \vdash P : \mathbf{T}(\mathsf{N}) \quad \Delta;\Gamma \vdash P' : \mathbf{T}(\mathsf{N})}{\Delta;\Gamma \vdash \mathsf{if}\ V\ \mathsf{then}\ P\ \mathsf{else}\ P'\ \mathsf{fi} : \mathbf{T}(\mathsf{N})}$$

$$[\text{ty-eq}]\ \frac{\Delta;\Gamma \vdash V : A\ \mathrm{ref} \quad \Delta;\Gamma \vdash V' : A\ \mathrm{ref}}{\Delta;\Gamma \vdash V = V' : \mathbf{T}(\mathsf{N})}$$

$$[\text{ty-alloc}]\ \frac{\Delta;\Gamma \vdash V : A}{\Delta;\Gamma \vdash \mathsf{ref}\ V : \mathbf{T}(A\ \mathrm{ref})} \qquad [\text{ty-deref}]\ \frac{\Delta;\Gamma \vdash V : A\ \mathrm{ref}}{\Delta;\Gamma \vdash !V : \mathbf{T}(A)}$$

$$[\text{ty-assign}]\ \frac{\Delta;\Gamma \vdash V : A\ \mathrm{ref} \quad \Delta;\Gamma \vdash V' : A}{\Delta;\Gamma \vdash V := V' : \mathbf{T}(\mathsf{U})}$$

$$[\text{ty-pred}]\ \frac{\Delta;\Gamma \vdash V : \mathsf{N}}{\Delta;\Gamma \vdash \mathsf{pred}\ V : \mathsf{N}} \qquad [\text{ty-succ}]\ \frac{\Delta;\Gamma \vdash V : \mathsf{N}}{\Delta;\Gamma \vdash \mathsf{succ}\ V : \mathsf{N}}$$

Fig. 3.1 – Règles de typage

Lemme 3.1.2

1. Pour tous Δ, Γ et G si $\Delta;\Gamma \vdash G : \gamma$, alors la dérivation de typage est unique.
2. Si $\Delta;\Gamma \vdash G : \gamma$, alors $locs(G) \subseteq dom(\Delta)$ et $fv(G) \subseteq dom(\Gamma)$.
3. Affaiblissement pour les états. Si $\Sigma : \Delta$, $\Delta' \subseteq \Delta$ et $\Sigma \subseteq \Sigma'$, alors $\Sigma' : \Delta'$.
4. Affaiblissement pour les termes. Si $\Delta;\Gamma \vdash G : \gamma$, alors $\Delta, \Delta';\Gamma, \Gamma' \vdash G : \gamma$.
5. Extension. Si $\Sigma : \Delta$, $\ell \notin locs(\Sigma)$ et $\Delta;\vdash v : \sigma$, alors $\Sigma[\ell \mapsto v] : \Delta, \ell : \sigma$.
6. Modification. Si $\Sigma : \Delta, \ell : \sigma$ et $\Delta;\vdash v : \sigma$ alors $\Sigma[\ell \mapsto v] : \Delta, \ell : \sigma$.
7. Substitution. Si $\Delta;\Gamma \vdash V : A$ et $\Delta;\Gamma, x : A \vdash M : \mathbf{T}(B)$ alors $\Delta;\Gamma \vdash M[V/x] : \mathbf{T}(B)$.

Démonstration : L'unicité de la dérivation est une conséquence de la présence de types sur les variables liées et du type d'état Δ. Les autres propriétés sont faciles. ■

3.1.2 Sémantique opérationnelle

Il existe plusieurs façons équivalentes de formuler la sémantique opérationnelle. La plus évidente est une relation d'évaluation à grands pas de la forme

$$\Sigma, M \Downarrow \Sigma', V$$

qui exprime qu'un terme fermé M, évalué dans l'état Σ, produit la valeur V et l'état (en général modifié) Σ'. Une autre possibilité, que nous avons choisie, est de définir un prédicat de terminaison sur les configurations, qui sont les paires d'un terme et d'une pile de contextes.[1] Dans notre langage, il est même possible d'exprimer les piles de contextes directement. L'avantage de ce style de sémantique par rapport à la sémantique à grands pas standard est que la définition du prédicat de terminaison est inductive sur la structure du terme, et évite la notion d'état intermédiaire du calcul, qui complique rapidement les preuves.

[1] La présentation par piles vient des contextes d'évaluation de Wright et Felleisen [WF94].

Si x est une variable, on définit l'ensemble $Cont_x$ des continuations en x par :

$$\frac{}{\mathsf{val}\ x \in Cont_x} \qquad \frac{fv(M) \subseteq \{x\} \quad K \in Cont_y}{\mathsf{let}\ y = M\ \mathsf{in}\ K \in Cont_x}$$

Le prédicat de terminaison prend alors la forme

$$\Sigma, \mathsf{let}\ x = M\ \mathsf{in}\ K \downarrow$$

où M est fermé et $K \in Cont_x$, ce qui signifie que, dans l'état Σ, l'évaluation de $\mathsf{let}\ x = M\ \mathsf{in}\ K$ termine. Les règles d'inférence qui définissent ce prédicat sont données en figure 3.2.

Il s'avère également pratique d'utiliser des continuations typées. On définit les jugements de typage $\Delta; \vdash K : (x : A)^\top$ par les règles suivantes :

$$\frac{}{\Delta; \vdash \mathsf{val}\ x : (x : A)^\top} \qquad \frac{\Delta; x : A \vdash M : \mathbf{T}(B) \quad \Delta; \vdash K : (y : B)^\top}{\Delta; \vdash \mathsf{let}\ y = M\ \mathsf{in}\ K : (x : A)^\top}$$

Lemme 3.1.3 (Équivariance de la terminaison)

Si $\pi : \mathbb{L} \to \mathbb{L}$ est une permutation alors

$$\Sigma, \mathsf{let}\ x = M\ \mathsf{in}\ K \downarrow \iff \pi \circ \Sigma \circ \pi^{-1}, \pi \bullet (\mathsf{let}\ x = M\ \mathsf{in}\ K) \downarrow$$

avec les définitions évidentes pour les actions du groupe des permutations sur les termes et les états.

Entre autres, le lemme précédent implique qu'on peut choisir n'importe quel nom pour les nouvelles références.

Proposition 3.1.4

Si $\ell, \ell' \notin locs(\Sigma) \cup locs(K) \cup locs(v)$, avec $v = in_{\mathbb{L}}\ell''$ ou $v = in_{\mathbb{Z}}n$, alors

$$\Sigma[\ell \mapsto v], \mathsf{let}\ x = \mathsf{val}\ \underline{\ell}\ \mathsf{in}\ K \downarrow \iff \Sigma[\ell' \mapsto v], \mathsf{let}\ x = \mathsf{val}\ \underline{\ell'}\ \mathsf{in}\ K \downarrow$$

Démonstration : Considérer la transposition $(\ell\, \ell')$. ■

Définition 3.1.5 (Contextes typés)

On définit les contextes $C[\cdot]$ comme des "termes avec un trou" et on écrit :

$$C[\cdot] : (\Delta; \Gamma \vdash \gamma) \Rightarrow (\Delta; - \vdash \mathbf{T}(A))$$

pour signifier que, quand $\Delta; \Gamma \vdash G : \gamma$, alors $\Delta; \vdash C[G] : \mathbf{T}(A)$.

$$[\text{op-term}]\ \frac{}{\Sigma, \mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ \mathsf{val}\ x\ \downarrow}$$

$$[\text{op-ret}]\ \frac{\Sigma, \mathsf{let}\ y = M[V/x]\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ (\mathsf{let}\ y = M\ \mathsf{in}\ K)\ \downarrow}$$

$$[\text{op-cc}]\ \frac{\Sigma, \mathsf{let}\ x_2 = M_1\ \mathsf{in}\ (\mathsf{let}\ x_1 = M_2\ \mathsf{in}\ K)\ \downarrow}{\Sigma, \mathsf{let}\ x_1 = (\mathsf{let}\ x_2 = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-app}]\ \frac{\Sigma, \mathsf{let}\ x_1 = M_1[V/x_2, (\mathsf{rec}\ f(x_2 : A_1) : A_2 = M_1)/f]\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x_1 = (\mathsf{rec}\ f(x_2 : A_1) : A_2 = M_1)\ V\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-succ}]\ \frac{\Sigma, \mathsf{let}\ x = \mathsf{val}\ \underline{n_1 + 1}\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \mathsf{succ}\ \underline{n_1}\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-pred}]\ \frac{\Sigma, \mathsf{let}\ x = \mathsf{val}\ \underline{n_1}\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \mathsf{pred}\ \underline{n_1 + 1}\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-if-true}]\ \frac{\Sigma, \mathsf{let}\ x = P\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \mathsf{if}\ \underline{0}\ \mathsf{then}\ P\ \mathsf{else}\ P'\ \mathsf{fi}\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-if-false}]\ \frac{\Sigma, \mathsf{let}\ x = P'\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \mathsf{if}\ \underline{n + 1}\ \mathsf{then}\ P\ \mathsf{else}\ P'\ \mathsf{fi}\ \mathsf{in}\ K\ \downarrow}$$

Fig. 3.2 – Sémantique opérationnelle

$$[\text{op-eq-true}]\ \frac{\Sigma, \mathsf{let}\ x = \mathsf{val}\ 0\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = (\ell = \ell)\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-eq-false}]\ \frac{\Sigma, \mathsf{let}\ x = \mathsf{val}\ 1\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = (\ell = \ell')\ \mathsf{in}\ K\ \downarrow}\ \ell \neq \ell'$$

$$[\text{op-assign-int}]\ \frac{\Sigma[l \mapsto in_{\mathbb{Z}} n], \mathsf{let}\ x = \mathsf{val}\ ()\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \underline{\ell} := \underline{n}\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-assign-loc}]\ \frac{\Sigma[l \mapsto in_{\mathbb{L}} \ell'], \mathsf{let}\ x = \mathsf{val}\ ()\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \underline{\ell} := \underline{\ell'}\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-deref-int}]\ \frac{\Sigma(l) = in_{\mathbb{Z}} n \quad \Sigma, \mathsf{let}\ x = \mathsf{val}\ \underline{n}\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = !\underline{\ell}\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-deref-loc}]\ \frac{\Sigma(l) = in_{\mathbb{L}} \ell' \quad \Sigma, \mathsf{let}\ x = \mathsf{val}\ \underline{\ell'}\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = !\underline{\ell}\ \mathsf{in}\ K\ \downarrow}$$

$$[\text{op-alloc-loc}]\ \frac{\Sigma[\ell \mapsto in_{\mathbb{L}} \ell'], \mathsf{let}\ x = \mathsf{val}\ \underline{\ell}\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \mathsf{ref}\ \underline{\ell'}\ \mathsf{in}\ K\ \downarrow}\ \ell \notin locs(\Sigma) \cup locs(K) \cup \{\ell'\}$$

$$[\text{op-alloc-int}]\ \frac{\Sigma[\ell \mapsto in_{\mathbb{Z}} n], \mathsf{let}\ x = \mathsf{val}\ \underline{\ell}\ \mathsf{in}\ K\ \downarrow}{\Sigma, \mathsf{let}\ x = \mathsf{ref}\ \underline{n}\ \mathsf{in}\ K\ \downarrow}\ \ell \notin locs(\Sigma) \cup locs(K) \cup \{\ell'\}$$

Fig. 3.3 – Sémantique opérationnelle (suite)

Définition 3.1.6 (Équivalence contextuelle)

Si $\Delta;\Gamma \vdash G_i : \gamma$ pour $i = 1, 2$ alors on écrit $\Delta;\Gamma \vdash G_1 \approx G_2 : \gamma$ pour

$$\forall A.\forall C[\cdot] : (\Delta;\Gamma \vdash \gamma) \Rightarrow (\Delta;- \vdash \mathbf{T}(A)).\forall \Sigma : \Delta.$$
$$\Sigma, \mathsf{let}\ x = C[G_1]\ \mathsf{in}\ \mathsf{val}\ x \downarrow \iff \Sigma, \mathsf{let}\ x = C[G_2]\ \mathsf{in}\ \mathsf{val}\ x \downarrow$$

3.2 Modèle de base

3.2.1 FM-cpos

On commence par introduire les définitions et résultats de la théorie des domaines de Fränkel-Mostowski nécessaires à la suite. Pour plus de détails, on pourra lire la thèse de Shinwell [Shi04].

On se donne un ensemble infini dénombrable $\mathbb{L}$ d'atomes (qui seront les adresses en mémoire). Un ensemble FM X est un ensemble muni d'une action du groupe des permutations de $\mathbb{L}$: une opération $\bullet$ des permutations de $\mathbb{L}$ dans les permutations de X telle que

$$\forall x \in X.\, id \bullet x = x$$
$$\forall \pi, \pi' \in perms(\mathbb{L}).\forall x \in X.\, \pi \bullet (\pi' \bullet x) = (\pi \circ \pi') \bullet x$$

et telle que chaque élément de X a un support fini : il existe un ensemble fini $L \subseteq \mathbb{L}$ tel que, pour toute permutation π qui laisse L invariant, $\pi \bullet x = x$. Si x a un support fini, il existe un plus petit ensemble L, que l'on note $supp(x)$.

Un morphisme entre ensembles FM est une fonction équivariante entre les ensembles sous-jacents :

$$\forall x.\forall \pi.\, \pi \bullet (f\, x) = f\, (\pi \bullet x)$$

Un FM-cpo est un ensemble FM muni d'une relation d'ordre $\sqsubseteq$ équivariante telle que les chaînes à support fini (c'est à dire dont les supports de tous les éléments sont inclus dans un même ensemble fini) ont une borne supérieure. Un morphisme de FM-cpo est un morphisme entre ensembles FM qui est monotone et qui préserve les bornes supérieures des chaînes à support fini. Il faut remarquer que, contrairement aux cpo usuels, certaines chaînes peuvent ne pas avoir de borne supérieure : si les éléments ont des supports croissants et non bornés, on ne trouvera pas de limite avec un support fini.

On note $\mathbb{Z}$, $\mathbb{B}$, etc. les ensembles correspondants, munis de l'action triviale (tous les éléments ont un support vide). $\mathbb{L}$ est l'ensemble des adresses en mémoire, vu comme un cpo discret avec l'action évidente : $\pi \bullet \ell = \pi(\ell)$.

La catégorie des FM-cpos est bicartésienne fermée : on écrit 1 et $\times$ pour les produits, $\Rightarrow$ pour la flèche et 0, + pour les coproduits. Les permutations agissent sur chaque composante des paires :

$$\pi \bullet (d, e) \;=\; (\pi \bullet d, \pi \bullet e)$$

et elles agissent par conjugaison sur les fonctions des objets $A \to B$:

$$\pi \bullet f \;=\; \lambda x.\pi \bullet (f\,(\pi^{-1} \bullet x))$$

Il faut noter que la catégorie n'est pas pointée : les morphismes de $1 \to D$ correspondent aux éléments de $1 \Rightarrow D$ à support vide, alors que les éléments des FM-cpos ont plus généralement un support fini.

L'opération de soulèvement est une monade, et la catégorie de Kleisli associée est celle des FM-cpo pointés et des fonctions continues strictes à support fini. Elle est symétrique monoïdale fermée, on note la flèche $\multimap$.

Si D est un FM cpo pointé, on définit $fix : (D \Rightarrow D) \multimap D$ comme d'habitude : la chaîne de$D$ a un support fini, car il est inclus dans celui de l'élément de $D \Rightarrow D$ qu'on donne a fix.

3.2.2 Définition de la sémantique

Sémantique des types

Le FM-cpo des états $\mathbb{S}$ est $\mathbb{L} \Rightarrow (\mathbb{Z}+\mathbb{L})$, c'est-à-dire les fonctions à support fini des adresses dans les valeurs enregistrables. Cet ensemble contient des fonctions avec une valeur par défaut (c'est à dire les fonctions constantes sauf sur un nombre fini d'adresses) mais aussi, par exemple, les fonctions qui sont l'identité partout sauf un nombre fini d'adresses.

On fait le lien avec les états finis de la sémantique opérationnelle en définissant

$$[\![\Sigma]\!] = \{S \in \mathbb{S} \mid \forall \ell \in dom\Sigma.S(\ell) = \Sigma(\ell)\}$$

Il faut noter que, quel que soit Σ, $[\![\Sigma]\!]$ n'est pas vide (il est même infini). Contrairement à ce qu'on aurait pu espérer, il n'y a pas de lien, ni d'inclusion, entre $locs(\Sigma)$ et $supp(S)$ pour $S \in [\![\Sigma]\!]$.

Le déréférencement est l'application, qui est continue et équivariante, comme la modification $\cdot[\cdot \mapsto \cdot]$ de type $\mathbb{S} \times \mathbb{L} \times (\mathbb{Z} + \mathbb{L}) \to \mathbb{S}$ définie par

$$(S[\ell \mapsto v])(\ell') \;=\; \begin{cases} v & \text{si } \ell = \ell' \\ S(\ell') & \text{sinon.} \end{cases}$$

La propriété suivante est facile à prouver :

$$\begin{aligned}
[\![\mathsf{U}]\!] &= 1 \\
[\![\mathsf{N}]\!] &= \mathbb{Z} \\
[\![\sigma \text{ ref}]\!] &= \mathbb{L} \\
[\![A \to \mathbf{T}(B)]\!] &= [\![A]\!] \Rightarrow \mathbf{T}([\![B]\!]) \\
\text{où} & \\
\mathbf{T}(D) &= (\mathbb{S} \Rightarrow D \Rightarrow \mathbb{O}) \multimap (\mathbb{S} \Rightarrow \mathbb{O})
\end{aligned}$$

Fig. 3.4 – Sémantique des types

Lemme 3.2.1

Pour tous Σ, ℓ, $v \in (\mathbb{Z} + \mathbb{L})$, $[\![\Sigma[\ell \to v]]\!] = \{S[\ell \to v] \mid S \in [\![\Sigma]\!]\}$.

L'équivalence des états de la sémantique opérationnelle est étendue naturellement aux états de la sémantique dénotationnelle. On obtient, comme prévu :

Lemme 3.2.2

Pour tout $\Sigma : \Delta$, et pour tous $S_1, S_2 \in [\![\Sigma]\!]$, $S_1 \sim S_2 : \Delta$.

On note $\mathbb{O}$ le FM-cpo plat $1_\perp$, qui contient deux éléments $\{\top, \perp\}$ avec l'action triviale. Le FM-cpo $[\![\gamma]\!]$ est défini inductivement par les règles données en Figure 3.4. L'interprétation des types de programmes T combine l'état et la continuation. L'utilisation d'une monade de continuations pour définir une sémantique élégante de la création de nom a été présentée par Shinwell et Pitts [SP05], dans le contexte de FreshML. Il faut aussi remarquer que les dénotations sont strictes : elles doivent utiliser la continuation.

Sémantique des jugements de typage

On définit la sémantique des contextes de typage par :

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!] = \{x_1 : [\![A_1]\!], \cdots, x_n : [\![A_n]\!]\}$$

C'est un enregistrement plutôt qu'une liste, car il est plus aisé d'utiliser des noms de variables : si $\rho \in [\![\Gamma]\!]$, on écrira $\rho(x)$ au lieu de $\pi_i(\rho)$.

Les jugements de typage sont interprétés par des éléments de l'ensemble sous-jacent de l'objet flèche interne, plutôt que par des morphismes de la catégorie :

$$[\![\Delta; \Gamma \vdash G : \gamma]\!] \in [\![\Gamma]\!] \Rightarrow [\![\gamma]\!]$$

et pas

$$[\![\Delta;\Gamma\vdash G:\gamma]\!]:[\![\Gamma]\!]\to[\![\gamma]\!]$$

En effet, la présence des adresses dans les programmes fait que leurs dénotations ont, en général, un support non vide.

Par ailleurs, les dénotations sont définies comme des fonctions sur tous les états, même si les théorèmes d'adéquation seront énoncés seulement pour les états du bon type. On utilisera aussi la notation $in_\sigma : [\![\sigma]\!] \to (\mathbb{Z}+\mathbb{L})$ pour tous les types enregistrables σ : $in_{\mathsf{N}}\, n = in_{\mathbb{Z}}\, n$ and $in_{\sigma\ \mathrm{ref}}\,\ell = in_{\mathbb{L}}\,\ell$.

La sémantique dénotationnelle est définie par induction sur la dérivation de typage (qui par le lemme 3.1.2, est unique). La plupart des clauses sont triviales, on ne rappelle en figure 3.5 que les cas intéressants.

Alors que la sémantique dénotationnelle est définie à l'aide de continuations, il n'y a pas d'opérateurs de contrôle dans notre langage, et, par conséquent, toutes les opérations sur lesdites continuations sont triviales. Elles ne servent qu'à définir quelles sont les adresses utilisées par le reste du programme, pour pouvoir en choisir une quand on doit interpréter la règle [ty-alloc] : les continuations explicites définissent par rapport à quoi cette adresse doit être nouvelle, la technique des FM-cpo nous assurant qu'il existe toujours une telle adresse, et que le choix n'importe pas. Pour le vérifier, on peut faire un calcul : en écrivant f pour

$$\lambda\ell.k\, S[\ell\mapsto in_\sigma([\![\Delta;\Gamma\vdash V:\sigma]\!]\rho)]\,\ell$$

et en choisissant $\ell_1,\ell_2\notin supp(f)$ distincts, alors

$$\begin{array}{lll}
& k\, S[\ell_1\mapsto in_\sigma([\![\Delta;\Gamma\vdash V:\sigma]\!]\rho)]\,\ell_1 & \\
= & f\,\ell_1 & \\
= & (\ell_1\ell_2)\bullet(f\,\ell_1) & \text{l'action sur } \mathbb{O} \text{ est triviale}\\
= & ((\ell_1\ell_2)\bullet f)\,((\ell_1\ell_2)\bullet\ell_1) & \text{action sur les fonctions}\\
= & f\,((\ell_1\ell_2)\bullet\ell_1) & \text{car } \ell_1,\ell_2\notin supp(f)\\
= & f\,\ell_2 & \text{transposition}\\
= & k\, S[\ell_2\mapsto in_\sigma([\![\Delta;\Gamma\vdash V:\sigma]\!]\rho)]\,\ell_2 &
\end{array}$$

Une autre possibilité, peut-être plus naturelle, de définir la sémantique de l'allocation est d'utiliser

"...pour n'importe quel $\ell\notin supp(k)\cup supp(S)\cup supp([\![\Delta;\Gamma\vdash V:\sigma]\!]\rho)$"

qui est équivalent, car

$$\begin{array}{ll}
& supp(\lambda\ell.k\, S[\ell\mapsto in_\sigma([\![\Delta;\Gamma\vdash V:\sigma]\!]\rho)]\,\ell)\\
\subseteq & supp(k)\cup supp(S)\cup supp([\![\Delta;\Gamma\vdash V:\sigma]\!]\rho)
\end{array}$$

[ty-loc] :

$$[\![\Delta;\Gamma \vdash \underline{\ell} : \sigma \text{ ref}]\!]\ \rho = \ell$$

[ty-let] :

$$[\![\Delta;\Gamma \vdash \mathsf{let}\ x = P\ \mathsf{in}\ P' : \mathbf{T}(B)]\!]\rho = \\ \lambda k.\lambda S.[\![\Delta;\Gamma \vdash P : \mathbf{T}(A)]\!]\ \rho\ (\lambda S'.\lambda d.[\![\Delta;\Gamma, x : A \vdash P' : \mathbf{T}(B)]\!]\ \rho[x \mapsto d]\ k\ S')\ S$$

[ty-val] :

$$[\![\Delta;\Gamma \vdash \mathsf{val}\ V : \mathbf{T}(A)]\!]\ \rho = \lambda k.\lambda S.k\ S\ ([\![\Delta;\Gamma \vdash V : A]\!]\ \rho)$$

[ty-deref] :

$$[\![\Delta;\Gamma \vdash !V : \mathbf{T}(\sigma)]\!]\ \rho = \lambda k.\lambda S.\begin{cases} k\ S\ x & \text{si } S([\![\Delta;\Gamma \vdash V : \sigma \text{ ref}]\!]\ \rho) = in_\sigma x \\ \bot & \text{sinon} \end{cases}$$

[ty-assign] :

$$[\![\Delta;\Gamma \vdash V := V' : \mathbf{T}(\text{unit})]\!]\ \rho = \\ \lambda k.\lambda S.k\ S[([\![\Delta;\Gamma \vdash V : \sigma \text{ ref}]\!]\ \rho)\ \mapsto\ in_\sigma([\![\Delta;\Gamma \vdash V' : \sigma]\!]\ \rho)] *$$

[ty-alloc] :

$$[\![\Delta;\Gamma \vdash \mathsf{ref}\ V : \mathbf{T}(\sigma \text{ ref})]\!]\rho = \lambda k.\lambda S.\, k\, S[\ell \mapsto in_\sigma([\![\Delta;\Gamma \vdash V : \sigma]\!]\rho)]\,\ell \\ \text{où } \ell \notin supp(\lambda \ell'.k\, S[\ell' \mapsto in_\sigma([\![\Delta;\Gamma \vdash V : \sigma]\!]\rho)]\,\ell')$$

[ty-rec] :

$$[\![\Delta;\Gamma \vdash (\mathsf{rec}\ f\ x = P) : A \to \mathbf{T}(B)]\!]\ \rho\ = \\ fix(\lambda\varphi.\lambda\chi.[\![\Delta;\Gamma, f : A \to \mathbf{T}(B), x : A \vdash P : \mathbf{T}(B)]\!]\ \rho[f \mapsto \varphi, x \mapsto \chi])$$

Fig. 3.5 – Sémantique des termes (extrait)

Nous avons choisi cette présentation pour montrer que la notion de support est sémantique, et pas syntaxique : la quantification couvre les adresses qui n'ont pas encore été utilisées, mais aussi celles qui ne sont plus accessibles depuis la continuation (garbage collector).

On définit aussi la sémantique des continuations si $\Delta;\vdash K : (x : A)^\top$ alors

$$\begin{array}{rcl}[\![\Delta;\vdash K : (x : A)^\top]\!]^K & = & \lambda S.\lambda d.[\![\Delta; x : A \vdash K : \mathbf{T}(B)]\!]\{x \mapsto d\}(\lambda S.\lambda d.\top)\, S \\ & : & \mathbb{S} \Rightarrow [\![A]\!] \Rightarrow \mathbb{O}\end{array}$$

Il est facile de vérifier ces quelques propriétés :

Lemme 3.2.3

1. La sémantique est bien définie.
2. $supp([\![\Delta;\Gamma \vdash G : \gamma]\!]) \subseteq locs(G)$.
3. Si $\Delta;\Gamma \vdash G : \gamma$ alors pour tout Δ', $[\![\Delta;\Gamma \vdash G : \gamma]\!] = [\![\Delta,\Delta';\Gamma \vdash G : \gamma]\!]$.
4. Substitution. Si $\Delta;\Gamma \vdash V : A$ et $\Delta;\Gamma, x : A \vdash P : \mathbf{T}(B)$ alors

   $$[\![\Delta;\Gamma \vdash P[V/x] : \mathbf{T}(B)]\!]\,\rho = [\![\Delta;\Gamma, x : A \vdash P : \mathbf{T}(B)]\!]\,\rho[x \mapsto [\![\Delta;\Gamma \vdash V : A]\!]]$$

5. Compositionalité. Si

   $$C[-] : (\Delta;\Gamma \vdash \gamma) \Rightarrow (\Delta;\vdash \mathbf{T}(A)) \quad \text{et} \quad [\![\Delta;\Gamma \vdash G_1 : \gamma]\!] = [\![\Delta;\Gamma \vdash G_2 : \gamma]\!]$$

   alors
   $$[\![\Delta;\vdash C[G_1] : \mathbf{T}(A)]\!] = [\![\Delta;\vdash C[G_2] : \mathbf{T}(A)]\!].$$

6. Termes et continuations :

   $$\begin{array}{l}[\![\Delta;\vdash \mathsf{let}\, y = M\, \mathsf{in}\, K : (x : A)^\top]\!]^K \\ \quad = \lambda S\lambda d.[\![\Delta; x : A \vdash M : \mathbf{T}(B)]\!]\,\{x \mapsto d\}\,[\![\Delta;\vdash K : (y : B)^\top]\!]^K\, S\end{array}$$

3.2.3 Correction et adéquation

Il s'agit maintenant de prouver la correction et l'adéquation de la sémantique dénotationnelle, par rapport à la sémantique opérationnelle.

Proposition 3.2.4 (Correction)

Si $\Delta;\vdash M : \mathbf{T}(A)$, $\Delta;\vdash K : (x : A)^\top$, $\Sigma : \Delta$ et $S \in [\![\Sigma]\!]$ alors
$\Sigma, \mathsf{let}\, x = M\, \mathsf{in}\, K \downarrow \implies [\![\Delta;\vdash M : \mathbf{T}(A)]\!]\{\}[\![\Delta;\vdash K : (x : A)^\top]\!]^K\, S = \top$

Démonstration : Par induction sur la preuve de termination. On donne seulement quelques cas intéressants :

Cas [op-term] :

$$
\begin{array}{ll}
& [\![\Delta; \vdash \mathsf{val}\ V : \mathbf{T}(A)]\!]\{\}[\![\Delta; \vdash \mathsf{val}\ x : (x : A)^\top]\!]^K\ S \\
= & \text{sémantique de val, définition de } [\![\cdot]\!]^K \\
& (\lambda k.\lambda S.k\ S\ ([\![\Delta; \vdash V : A]\!]\{\}))\ (\lambda S\lambda d.[\![\Delta; x : A \vdash \mathsf{val}\ x : \mathbf{T}(A)]\!]\ \{x \mapsto d\}\ (\lambda S.\lambda d.\top)\ S)\ S \\
= & \text{réduction} \\
& (\lambda S\lambda d.[\![\Delta; x : A \vdash \mathsf{val}\ x : \mathbf{T}(A)]\!]\ \{x \mapsto d\}\ (\lambda S.\lambda d.\top)\ S)\ S\ ([\![\Delta; \vdash V : A]\!]\{\}) \\
= & \text{sémantique de val et } x\text{, réduction} \\
& (\lambda S\lambda d.\top)\ S\ ([\![\Delta; \vdash V : A]\!]\{\}) \\
= & \text{réduction} \\
& \top
\end{array}
$$

Cas [op-ret] : Par hypothèse, on a

$$
\begin{array}{c}
\Sigma, \mathsf{let}\ y = M[V/x]\ \mathsf{in}\ K\ \downarrow \\
\Delta; \vdash \mathsf{val}\ V : \mathbf{T}(A) \\
\Delta; \vdash \mathsf{let}\ y = M\ \mathsf{in}\ K : (x : A)^\top \\
\Sigma : \Delta \\
S \in [\![\Sigma]\!]
\end{array}
$$

Ainsi, par la règle de typage pour les continuations, pour un certain B,

$$
\begin{array}{c}
\Delta; x : A \vdash M : \mathbf{T}(B) \\
\Delta; \vdash K : (y : B)^\top
\end{array}
$$

Par le lemme 3.1.2 (substitution) on a

$$\Delta; \vdash M[V/x] : \mathbf{T}(B)$$

et donc on peut appliquer l'hypothèse d'induction, et on obtient

$$[\![\Delta; \vdash M[V/x] : \mathbf{T}(B)]\!]\ \{\}\ [\![\Delta; \vdash K : (y : B)^\top]\!]^K\ S = \top$$

On peut alors calculer

$$
\begin{array}{ll}
& [\![\Delta; \vdash M[V/x] : \mathbf{T}(B)]\!]\ \{\}\ [\![\Delta; \vdash K : (y : B)^\top]\!]^K\ S \\
= & \text{lemme 3.2.3 (substitution)} \\
& [\![\Delta; x : A \vdash M : \mathbf{T}(B)]\!]\{x \mapsto [\![\Delta; \vdash V : A]\!]\{\}\}\ [\![\Delta; \vdash K : (y : B)^\top]\!]^K\ S \\
= & \text{lemme 3.2.3 (termes et continuations)} \\
& [\![\Delta; \vdash \mathsf{let}\ y = M\ \mathsf{in}\ K : (x : A)^\top]\!]^K\ S\ ([\![\Delta; \vdash V : A]\!]\ \{\}) \\
= & \text{sémantique de val} \\
& [\![\Delta; \vdash \mathsf{val}\ V : \mathbf{T}(A)]\!]\ \{\}\ [\![\Delta; \vdash \mathsf{let}\ y = M\ \mathsf{in}\ K : (x : A)^\top]\!]^K\ S
\end{array}
$$

Cas [op-alloc-loc] : Par hypothèse, en examinant les règles [ty-loc] et [ty-alloc], on a Σ, Δ, K, σ, S, ℓ tels que

$$\begin{array}{l}\Sigma : \Delta, \ell' : \sigma \\ S \in [\![\Sigma]\!] \\ \Delta, \ell' : \sigma; \vdash \mathsf{ref}\ \underline{\ell'} : \mathbf{T}((\sigma\ \mathrm{ref}\ \mathrm{ref})) \\ \Delta, \ell' : \sigma; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top} \\ \ell \notin locs(\Sigma) \cup locs(K) \cup \{\ell'\} \\ \Sigma[\ell \mapsto in_{\mathbb{L}}\ell'], \mathsf{let}\ x{=}\mathsf{val}\ \underline{\ell}\ \mathsf{in}\ K\ \downarrow\end{array}$$

et on veut prouver que

$$[\![\Delta, \ell' : \sigma; \vdash \mathsf{ref}\ \underline{\ell'} : \mathbf{T}(\sigma\ \mathrm{ref}\ \mathrm{ref})]\!]\ \{\}\ [\![\Delta, \ell' : \sigma; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K\ S\ = \top$$

Par la proposition 3.1.4, on peut aussi supposer que

$$\ell \notin supp(S) \cup supp([\![\Delta, \ell' : \sigma; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K)$$

Par le lemme 3.1.2 (extension)

$$\Sigma[\ell \mapsto in_{\mathbb{L}}(\ell')] : \Delta, \ell' : \sigma, \ell : \sigma\ \mathrm{ref}$$

et par le lemme 3.2.1

$$S[\ell \mapsto in_{\mathbb{L}}(\ell')] \in [\![\Sigma[\ell \mapsto in_{\mathbb{L}}(\ell')]]\!]$$

donc on peut appliquer l'hypothèse d'induction pour obtenir

$$[\![\Delta, \ell', \ell; \vdash \mathsf{val}\ \underline{\ell}]\!]\ \{\}\ [\![\Delta, \ell', \ell; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K\ S[\ell \mapsto in_{\mathbb{L}}(\ell')] = \top$$

On peut alors enrichir le contexte pour K et faire un calcul

$$\begin{array}{ll} & [\![\Delta, \ell', \ell; \vdash \mathsf{val}\ \underline{\ell} : \mathbf{T}(\sigma\ \mathrm{ref}\ \mathrm{ref})]\!]\ \{\}\ [\![\Delta, \ell'; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K\ S[\ell \mapsto in_{\mathbb{L}}(\ell')] \\ = & \text{sémantique de val} \\ & [\![\Delta, \ell'; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K\ S[\ell \mapsto in_{\mathbb{L}}(\ell')]\ ([\![\Delta, \ell', \ell; \vdash \underline{\ell} : \sigma\ \mathrm{ref}\ \mathrm{ref}]\!]\ \{\}) \\ = & \text{sémantique de } \underline{\ell} \\ & [\![\Delta, \ell'; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K\ S[\ell \mapsto in_{\mathbb{L}}(\ell')]\ \ell \\ = & \text{sémantique de } \underline{\ell'} \\ & [\![\Delta, \ell'; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K\ S[\ell \mapsto in_{\sigma\ \mathrm{ref}}([\![\Delta, \ell'; \vdash \underline{\ell'} : \sigma\ \mathrm{ref}]\!]\ \{\})]\ \ell \\ = & \text{sémantique de l'allocation, hypothèses sur } \ell \\ & [\![\Delta, \ell'; \vdash \mathsf{ref}\ \underline{\ell'} : \mathbf{T}(\sigma\ \mathrm{ref}\ \mathrm{ref})]\!]\ \{\}\ [\![\Delta, \ell'; \vdash K : (x : \sigma\ \mathrm{ref}\ \mathrm{ref})^{\top}]\!]^K\ S \end{array}$$

comme attendu. ■

Pour prouver l'autre direction, on utilise des relations logiques d'approximation formelle entre la syntaxe et la sémantique :

$$
\begin{array}{rcl}
\mathbb{R}^{\Delta}_{\mathbb{S}} & = & \{(\Sigma, S) \mid \Sigma : \Delta \wedge S \in [\![\Sigma]\!]\} \\
\mathbb{R}^{\Delta}_{\mathsf{U}} & = & \{((), *)\} \\
\mathbb{R}^{\Delta}_{\mathsf{N}} & = & \{(\underline{n}, n) \mid n \in \mathbb{Z}\} \\
\mathbb{R}^{\Delta}_{\sigma \text{ ref}} & = & \{(\underline{\ell}, \ell) \mid \Delta; \vdash \underline{\ell} : \sigma \text{ ref}\} \\
\mathbb{R}^{\Delta}_{A_1 \to \mathbf{T}(A_2)} & = & \{(W, g) \mid \Delta; \vdash W : A_1 \to \mathbf{T}(A_2) \wedge g \in [\![A_1 \to \mathbf{T}(A_2)]\!] \wedge \forall \Delta' \supseteq \Delta. \\
 & & \quad \forall (V, d) \in \mathbb{R}^{\Delta'}_{A_1}.(W\ V, g\ d) \in \mathbb{R}^{\Delta'}_{\mathbf{T}(A_2)}\} \\
\mathbb{R}^{\Delta}_{x:A^{\top}} & = & \{(K, k) \mid \Delta; \vdash K : (x : A)^{\top} \wedge k : \mathbb{S} \Rightarrow [\![A]\!] \Rightarrow \mathbb{O} \wedge \forall \Delta' \supseteq \Delta. \\
 & & \quad \forall (V, d) \in \mathbb{R}^{\Delta'}_{A}.\forall (\Sigma, S) \in \mathbb{R}^{\Delta'}_{\mathbb{S}}.k\ S\ d = \top \implies \Sigma, \mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ K\ \downarrow\} \\
\mathbb{R}^{\Delta}_{\mathbf{T}(A)} & = & \{(M, d) \mid \Delta; \vdash M : \mathbf{T}(A) \wedge d \in [\![\mathbf{T}(A)]\!] \wedge \forall \Delta' \supseteq \Delta. \\
 & & \quad \forall (K, k) \in \mathbb{R}^{\Delta'}_{x:A^{\top}}.\forall (\Sigma, S) \in \mathbb{R}^{\Delta'}_{\mathbb{S}}.d\ k\ S = \top \implies \Sigma, \mathsf{let}\ x = M\ \mathsf{in}\ K\ \downarrow\}
\end{array}
$$

On les étend aux termes ouverts :

$$
\begin{array}{rcl}
\mathbb{R}^{\Delta}_{\vec{x_i}:\vec{A_i} \vdash A} & = & \{(V, f) \mid \Delta; \vec{x_i} : \vec{A_i} \vdash V : A \wedge f \in [\![\vec{x_i} : \vec{A_i}]\!] \Rightarrow [\![A]\!] \wedge \forall \Delta' \supseteq \Delta \\
 & & \quad \forall (\vec{V_i}, \vec{d_i}) \in \mathbb{R}^{\Delta'}_{\vec{A_i}}.\ (V[V_i/x_i], f\ \{x_i \mapsto d_i\}) \in \mathbb{R}^{\Delta'}_{A}\} \\
\mathbb{R}^{\Delta}_{\vec{x_i}:\vec{A_i} \vdash \mathbf{T}(A)} & = & \{(M, f) \mid \Delta; \vec{x_i} : \vec{A_i} \vdash M : \mathbf{T}(A) \wedge f \in [\![\vec{x_i} : \vec{A_i}]\!] \Rightarrow \mathbf{T}([\![A]\!]) \wedge \forall \Delta' \supseteq \Delta \\
 & & \quad \forall (\vec{V_i}, \vec{d_i}) \in \mathbb{R}^{\Delta'}_{\vec{A_i}}.\ (M[V_i/x_i], f\ \{x_i \mapsto d_i\}) \in \mathbb{R}^{\Delta'}_{\mathbf{T}(A)}\}
\end{array}
$$

Le lemme suivant établit quelques propriétés élémentaires des relations :

Lemme 3.2.5

1. Si $\Delta' \supseteq \Delta$ alors $\mathbb{R}^{\Delta'}_{\gamma} \supseteq \mathbb{R}^{\Delta}_{\gamma}$ (resp. pour les termes ouverts, les continuations)
2. Si $(M, f) \in \mathbb{R}^{\Delta}_{x:A_1 \vdash \mathbf{T}(A_2)}$ et $(K, k) \in \mathbb{R}^{\Delta}_{y:A_2^{\top}}$ alors
$$(\mathsf{let}\ y = M\ \mathsf{in}\ K, \lambda S d.f\ \{x \mapsto d\}\ k\ S) \in \mathbb{R}^{\Delta}_{x:A_1^{\top}}.$$
3. $(\mathsf{val}\ x, \lambda S.\lambda d.\top) \in \mathbb{R}^{\Delta}_{x:A^{\top}}$.
4. Si $\Delta; \vdash F : A \to \mathbf{T}(B)$ alors $(F, \lambda d.\lambda k.\lambda S.\bot) \in \mathbb{R}^{\Delta}_{A \to \mathbf{T}(B)}$.
5. Si $f \sqsubseteq g$ et $(M, g) \in \mathbb{R}^{\Delta}_{\mathbf{T}(A)}$, alors $(M, f) \in \mathbb{R}^{\Delta}_{\mathbf{T}(A)}$.
6. Si $f_0 \sqsubseteq f_1 \sqsubseteq \ldots$ est une chaîne à support fini de $[\![\mathbf{T}(A)]\!]$ et si pour tout n, $(M, f_n) \in \mathbb{R}^{\Delta}_{\mathbf{T}(A)}$, alors $(M, \bigsqcup f_n) \in \mathbb{R}^{\Delta}_{\mathbf{T}(A)}$.

Démonstration :

1. Découle immédiatement des définitions.
2. Notons que $\Delta;\vdash \mathsf{let} y = M \mathsf{in} K : (x : A_1)^\top$. Soient $\Delta' \supseteq \Delta$, $(V,d) \in \mathbb{R}^{\Delta'}_{A_1}$ et $(\Sigma, S) \in \mathbb{R}^{\Delta'}_{\mathbb{S}}$ tels que $(\lambda S d.f\ \{x \mapsto d\}\ k\ S)\, S\, d = \top$. Comme $(V,d) \in \mathbb{R}^{\Delta'}_{A_1}$, $(M[V/x], f\{x \mapsto d\}) \in \mathbb{R}^{\Delta'}_{\mathbf{T}(A)_2}$. Puisque $(K,k) \in \mathbb{R}^{\Delta'}_{y:A_2^\top}$ et $(\Sigma, S) \in \mathbb{R}^{\Delta'}_{\mathbb{S}}$, $f\{x \mapsto d\}\, k\, S = \top$ implique que $\Sigma, \mathsf{let}\ y = M[V/x]\ \mathsf{in}\ K \downarrow$, ce qui amène

$$\Sigma, \mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ K \downarrow .$$

3. Pour tous Σ, V, $\Sigma, \mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ \mathsf{val}\ x \downarrow$.
4. Pour tout F,V,K,Σ, on a $(\bot = \top) \implies \Sigma, \mathsf{let}\ x = F\ V\ \mathsf{in}\ K \downarrow$.
5. Supposons que $f \sqsubseteq g$ et $(M,g) \in \mathbb{R}^{\Delta}_{\mathbf{T}(A)}$. Soient $\Delta' \supseteq \Delta$, $(\Sigma, S) \in \mathbb{R}^{\Delta'}_{\mathbb{S}}$ et $(K,k) \in \mathbb{R}^{\Delta'}_{x:A^\top}$ tels que $f\, k\, S = \top$. Comme $f \sqsubseteq g$, $g\, k\, S = \top$ également, $\Sigma, \mathsf{let}\ x = M\ \mathsf{in}\ K \downarrow$.
6. Soit f_n une chaîne à support fini telle que, pour tout n, $(M, f_n) \in \mathbb{R}^{\Delta}_{\mathbf{T}(A)}$. Soient $\Delta' \supseteq \Delta$, $(\Sigma, S) \in \mathbb{R}^{\Delta'}_{\mathbb{S}}$ et $(K,k) \in \mathbb{R}^{\Delta'}_{x:A^\top}$ tels que $\bigsqcup f_n\, k\, S = \top$. Par continuité il existe n tel que $f_n\, k\, S = \top$, donc $\Sigma, \mathsf{let}\ x = M\ \mathsf{in}\ K \downarrow$. ■

Lemme 3.2.6 (Lemme Fondamental)

1. Si $\Delta;\Gamma \vdash V : A$ alors $(V, [\![\Delta;\Gamma \vdash V : A]\!]) \in \mathbb{R}^{\Delta}_{\Gamma\vdash A}$.
2. Si $\Delta;\Gamma \vdash M : \mathbf{T}(A)$ alors $(M, [\![\Delta;\Gamma \vdash M : \mathbf{T}(A)]\!]) \in \mathbb{R}^{\Delta}_{\Gamma\vdash \mathbf{T}(A)}$.
3. Si $\Delta;\vdash K : (x : A)^\top$, alors $(K, [\![\Delta;\vdash K : (x : A)^\top]\!]^K) \in \mathbb{R}^{\Delta}_{x:A^\top}$.

Démonstration : Pour les valeurs comme les programmes, on fait une induction sur la dérivation de typage. On prouve seulement quelques cas. Soit $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ et supposons que $\Delta' \supseteq \Delta$, $(V_i, d_i) \in \mathbb{R}^{\Delta'}_{A_i}$ et que $\rho = \{x_i \mapsto d_i\}$.

Cas [ty-var] : Le couple $(x_j[V_i/x_i], [\![\Delta;\Gamma \vdash x_j : A_j]\!]\, \rho)$ est (V_j, d_j), qui est dans $\mathbb{R}^{\Delta'}_{A_j}$ par hypothèse.

Cas [ty-int] : $(\underline{n}[V_i/x_i], [\![\Delta;\Gamma \vdash \underline{n} : \mathsf{N}]\!]) = (\underline{n}, n) \in \mathbb{R}^{\Delta'}_{\mathsf{N}}$ par définition.

Cas [ty-rec] :

$$\begin{array}{ll} & [\![\Delta;\Gamma\vdash(\mathsf{rec}\ f\ x=P):A\to\mathbf{T}(B)]\!]\ \rho\\ = & fix(\lambda\varphi:[\![A\to\mathbf{T}(B)]\!].\lambda\chi:[\![A]\!].[\![\Delta;\Gamma,f,x\vdash P:\mathbf{T}(B)]\!]\ \rho[f\mapsto\varphi,x\mapsto\chi])\\ = & \bigsqcup f_i\end{array}$$

où

$$\begin{array}{rcl} f_0 & = & \lambda\chi.\lambda k.\lambda S.\bot\\ f_{i+1} & = & \lambda\chi.[\![\Delta;\Gamma,f:A\to\mathbf{T}(B),x:A\vdash P:\mathbf{T}(B)]\!]\ \rho[f\mapsto f_i,x\mapsto\chi])\end{array}$$

Par le lemme 3.2.5, $(\mathsf{rec}\, f\, x = P[V_i/x_i], f_0) \in \mathbb{R}^{\Delta'}_{A\to\mathbf{T}(B)}$. Par hypothèse d'induction, on obtient le même résultat pour chaque f_i. Ainsi, le lemme 3.2.5 nous dit que pour tout V, $\{f \mid (V,f) \in \mathbb{R}^{\Delta}_{A\to\mathbf{T}(B)}\}$ est clos par limite de chaîne, ce qui conclut la preuve.

Cas [ty-loc] : Immédiat.

Cas [ty-app] : Par induction

$$\begin{array}{l}(V[V_i/x_i],[\![\Delta;\Gamma\vdash V:A\to\mathbf{T}(B)]\!]\rho)\in\mathbb{R}^{\Delta'}_{A\to\mathbf{T}(B)} \qquad \text{et}\\ (V'[V_i/x_i],[\![\Delta;\Gamma\vdash V':A]\!]\rho)\in\mathbb{R}^{\Delta'}_{A}.\end{array}$$

Ainsi

$$\begin{array}{ll} & ((V\ V')[V_i/x_i],[\![\Delta;\Gamma\vdash V\ V':\mathbf{T}(B)]\!]\rho)\\ = & (V[V_i/x_i]\ V'[V_i/x_i],([\![\Delta;\Gamma\vdash V:A\to\mathbf{T}(B)]\!]\rho)\ ([\![\Delta;\Gamma\vdash V':A]\!]\rho))\\ \in & \mathbb{R}^{\Delta'}_{\mathbf{T}(B)}\end{array}$$

Cas [ty-let] : On veut prouver

$$((\mathsf{let}\ x=M_1\ \mathsf{in}\ M_2)[V_i/x_i],[\![\Delta;\Gamma\vdash\mathsf{let}\ x=M_1\ \mathsf{in}\ M_2:\mathbf{T}(A_2)]\!]\ \rho)\in\mathbb{R}^{\Delta'}_{\mathbf{T}(A_2)}$$

Soient $\Delta'' \supseteq \Delta'$, $(K,k) \in \mathbb{R}^{\Delta''}_{y:A_2^\top}$ et $(\Sigma,S) \in \mathbb{R}^{\Delta''}_{\mathbb{S}}$. Par induction, on peut prouver que

$$(M_2[V_i/x_i],\lambda\{x\mapsto d\}.[\![\Delta;\Gamma\vdash M_2:\mathbf{T}(A_2)]\!]\ \rho[x\mapsto d])\in\mathbb{R}^{\Delta'}_{x:A_1\vdash\mathbf{T}(A_2)}$$

donc, par le lemme 3.2.5 point 2,

$$(\mathsf{let}\ y=M_2[V_i/x_i]\ \mathsf{in}\ K,\lambda Sd.[\![\Delta;\Gamma\vdash M_2:\mathbf{T}(A_2)]\!]\ \rho[x\mapsto d]\ k\ S)\in\mathbb{R}^{\Delta''}_{x:A_1^\top}$$

Encore une fois par induction

$$(M_1[V_i/x_i],[\![\Delta;\Gamma\vdash M_1:\mathbf{T}(A_1)]\!]\ \rho)\in\mathbb{R}^{\Delta'}_{\mathbf{T}(A_1)}$$

donc

$$\begin{array}{c}[\![\Delta;\Gamma\vdash M_1:\mathbf{T}(A_1)]\!]\,\rho\,(\lambda Sd.[\![\Delta;\Gamma\vdash M_2:\mathbf{T}(A_2)]\!]\,\rho[x\mapsto d]\,k\,S)S \;=\; \top\\ \Longrightarrow\\ \Sigma,\mathsf{let}\;x=M_1[V_i/x_i]\;\mathsf{in}\;(\mathsf{let}\;y=M_2[V_i/x_i]\;\mathsf{in}\;K)\;\downarrow\end{array}$$

Puisque le premier membre de l'égalité est exactement

$$[\![\Delta;\Gamma\vdash \mathsf{let}\;x=M_1\;\mathsf{in}\;M_2:\mathbf{T}(A_2)]\!]\,\rho\,k\,S \;=\; \top$$

et que le dernier implique que

$$\Sigma,\mathsf{let}\;y=(\mathsf{let}\;x=M_1\;\mathsf{in}\;M_2)[V_i/x_i]\;\mathsf{in}\;K\;\downarrow$$

la preuve est terminée.

Cas [ty-val] : On veut prouver que

$$((\mathsf{val}\;V)[V_i/x_i],[\![\Delta;\Gamma\vdash \mathsf{val}\;V:\mathbf{T}(A)]\!]\,\rho)\in\mathbb{R}^{\Delta'}_{\mathbf{T}(A)}$$

Soient $\Delta''\supseteq\Delta'$, $(K,k)\in\mathbb{R}^{\Delta''}_{x:A^\top}$ et $(\Sigma,S)\in\mathbb{R}^{\Delta''}_{\mathbb{S}}$. Par induction

$$(V[V_i/x_i],[\![\Delta;\Gamma\vdash V:A]\!]\,\rho)\in\mathbb{R}^{\Delta'}_{A}$$

Donc

$$\begin{array}{c}k\,S\,([\![\Delta;\Gamma\vdash V:A]\!]\,\rho) \;=\; \top\\ \Longrightarrow\\ \Sigma,\mathsf{let}\;x=\mathsf{val}\;(V[V_i/x_i])\;\mathsf{in}\;K\;\downarrow\end{array}$$

Puisque le premier membre de l'égalité est exactement

$$[\![\Delta;\Gamma\vdash \mathsf{val}\;V:\mathbf{T}(A)]\!]\,\rho\,k\,S \;=\; \top$$

et que le dernier est évidemment

$$\Sigma,\mathsf{let}\;x=(\mathsf{val}\;V)[V_i/x_i]\;\mathsf{in}\;K\;\downarrow$$

la preuve est terminée.

Cas [ty-deref] : On veut prouver que

$$(!V[V_i/x_i],[\![\Delta;\Gamma\vdash !V:\mathbf{T}(\sigma)]\!]\,\rho)\in\mathbb{R}^{\Delta'}_{\mathbf{T}(\sigma)}$$

Par [ty-loc], il faut $V[V_i/x_i]=\underline{\ell}$ pour une adresse ℓ telle que $\ell:\sigma\in\Delta'$, et par induction

$$(V[V_i/x_i],[\![\Delta;\Gamma\vdash V:\sigma\;\mathrm{ref}]\!]\,\rho)\in\mathbb{R}^{\Delta'}_{\sigma\;\mathrm{ref}}$$

ce qui implique que $[\![\Delta;\Gamma \vdash V : \sigma\ \mathrm{ref}]\!]\,\rho = \ell$ également. Soient $\Delta'' \supseteq \Delta'$, $(K,k) \in \mathbb{R}^{\Delta''}_{x:\sigma^\top}$ et $(\Sigma,S) \in \mathbb{R}^{\Delta''}_{\mathbb{S}}$. Puisque $\Sigma : \Delta'' \supseteq \Delta' \ni (l:\sigma)$ et $S \in [\![\Sigma]\!]$, il existe un v tel que $\Sigma(\ell) = in_\sigma v$ et $S(\ell) = in_\sigma v$. Donc, en développant la sémantique de la lecture en mémoire, on voit que prouver

$$[\![\Delta;\Gamma \vdash !V : \mathbf{T}(\sigma)]\!]\,\rho\,k\,S \;=\; \top \implies \Sigma, \mathsf{let}\ x = !V[V_i/x_i]\ \mathsf{in}\ K \downarrow$$

revient à prouver

$$k\,S\,v \;=\; \top \quad\implies\quad \Sigma, \mathsf{let}\ x = !\underline{\ell}\ \mathsf{in}\ K \downarrow$$

qu'on obtient, par la sémantique opérationnelle ([op-derefn], [op-derefl]) si on prouve que

$$k\,S\,v \;=\; \top \quad\implies\quad \Sigma, \mathsf{let}\ x = \mathsf{val}\ \underline{v}\ \mathsf{in}\ K \downarrow$$

Comme $(K,k) \in \mathbb{R}^{\Delta''}_{x:\sigma^\top}$, l'implication est vraie s'il existe un $\Delta''' \supseteq \Delta''$ tel que $(\underline{v},v) \in \mathbb{R}^{\Delta'''}_{\sigma}$ et $(\Sigma,S) \in \mathbb{R}^{\Delta'''}_{\mathbb{S}}$. Si $\sigma = \mathsf{N}$, et donc $v = n$ pour un $n \in \mathbb{Z}$, alors on choisit $\Delta''' = \Delta''$. Si $\sigma = \sigma'\,\mathrm{ref}$ pour un σ', et donc $v = \ell'$ pour une adresse $\ell' \in \mathbb{L}$, alors on considère deux cas. D'abord, si $\ell' \in dom(\Delta'')$ alors on peut prendre $\Delta''' = \Delta''$ et la preuve est terminée. Sinon $\ell' \notin dom(\Delta'')$, et on choisit alors $\Delta''' = \Delta'', \ell' : \sigma'$. Par définition du typage des états, on doit avoir $\Sigma \sim \Sigma : (\ell' : \sigma')$ et par définition de $[\![\Sigma]\!]$ cela signifie que $(\Sigma,S) \in \mathbb{R}^{\Delta'''}_{\mathbb{S}}$ donc la preuve est terminée.

Cas [ty-alloc] : On veut prouver

$$((\mathsf{ref}\ V)[V_i/x_i], [\![\Delta;\Gamma \vdash \mathsf{ref}\ V : \mathbf{T}(\sigma\ \mathrm{ref})]\!]\,\rho) \in \mathbb{R}^{\Delta'}_{\mathbf{T}(\sigma\ \mathrm{ref})}$$

Soient $\Delta'' \supseteq \Delta'$, $(K,k) \in \mathbb{R}^{\Delta''}_{x:\sigma\ \mathrm{ref}^\top}$ et $(\Sigma,S) \in \mathbb{R}^{\Delta''}_{\mathbb{S}}$. Comme, par induction

$$(V[V_i/x_i], [\![\Delta;\Gamma \vdash V : \sigma]\!]\,\rho) \in \mathbb{R}^{\Delta''}_{\sigma}$$

il existe un $v \in [\![\sigma]\!]$ tel que $[\![\Delta;\Gamma \vdash V : \sigma]\!]\,\rho = v$ et $V[V_i/x_i] = \underline{v}$. On choisit

$$\ell \notin supp(\lambda\ell'.k\,S[\ell' \mapsto in_\sigma([\![\Delta;\Gamma \vdash V : \sigma]\!]\rho)]\,\ell') \cup locs(\Sigma) \cup locs(K) \cup locs(V[V_i/x_i])$$

et on constate alors (lemme 3.2.1 et typage d'un état) que

$$(\Sigma[\ell \mapsto in_\sigma \underline{v}], S[\ell \mapsto in_\sigma v]) \in \mathbb{R}^{\Delta'',\ell:\sigma}_{\mathbb{S}}$$

et $(\underline{\ell},\ell) \in \mathbb{R}^{\Delta'',\ell:\sigma}_{\sigma\ \mathrm{ref}}$, ce qui nous amène au raisonnement suivant

$$\begin{array}{ll}
 & [\![\Delta;\Gamma \vdash \mathsf{ref}\ V : \mathbf{T}(\sigma\ \mathrm{ref})]\!]\,\rho\,k\,S \;=\; \top \\
\implies & \text{sémantique de l'allocation, fraîcheur de } \ell \\
 & k\,S[\ell \mapsto in_\sigma([\![\Delta;\Gamma \vdash V : \sigma]\!]\,\rho)]\,\ell \;=\; \top \\
\implies & k\,S[\ell \mapsto in_\sigma v]\,\ell \;=\; \top \\
\implies & \text{par hypothèse sur } (K,k) \\
 & \Sigma[\ell \mapsto in_\sigma \underline{v}], \mathsf{let}\ x = \mathsf{val}\ \underline{\ell}\ \mathsf{in}\ K \downarrow \\
\implies & \Sigma[\ell \mapsto in_\sigma(V[V_i/x_i])], \mathsf{let}\ x = \mathsf{val}\ \underline{\ell}\ \mathsf{in}\ K \downarrow \\
\implies & \text{par [op-allocl] ou [op-allocn] et fraîcheur de } \ell \\
 & \Sigma, \mathsf{let}\ x = \mathsf{ref}\ V[V_i/x_i]\ \mathsf{in}\ K \downarrow
\end{array}$$

comme attendu.

Cas [ty-assign] : Par induction

$$\begin{array}{rcl}(V[V_i/x_i], [\![\Delta; \Gamma \vdash V : \sigma \text{ ref}]\!]\ \rho) & \in & \mathbb{R}^{\Delta'}_{\sigma \text{ ref}} \\ (V'[V_i/x_i], [\![\Delta; \Gamma \vdash V' : \sigma]\!]\ \rho) & \in & \mathbb{R}^{\Delta'}_{\sigma}\end{array}$$

Donc, pour certains ℓ et v,

$$\begin{array}{l}V[V_i/x_i] = \underline{\ell} \quad \text{et} \quad [\![\Delta; \Gamma \vdash V : \sigma \text{ ref}]\!]\ \rho = \ell \\ V'[V_i/x_i] = \underline{v} \quad \text{et} \quad [\![\Delta; \Gamma \vdash V' : \sigma]\!]\ \rho = v\end{array}$$

Par conséquent, si $\Delta'' \supseteq \Delta'$, $(K, k) \in \mathbb{R}^{\Delta''}_{x:\text{unit}^{\top}}$ et $(\Sigma, S) \in \mathbb{R}^{\Delta''}_{\mathbb{S}}$

$$\begin{array}{rl} & [\![\Delta; \Gamma \vdash V := V' : \mathbf{T}(\text{unit})]\!]\ \rho\ k\ S = \top \\ \equiv & k\ S[\ell \mapsto in_\sigma v]\ * = \top \\ \Longrightarrow & \Sigma[\ell \mapsto in_\sigma \underline{v}], \mathsf{let}\ x = \mathsf{val}\ ()\ \mathsf{in}\ K\ \downarrow \\ & \quad \text{car } ((), *) \in \mathbb{R}^{\Delta''}_{\text{unit}} \text{ et } (\Sigma[\ell \mapsto in_\sigma \underline{v}], S[\ell \mapsto in_\sigma v]) \in \mathbb{R}^{\Delta''}_{\mathbb{S}} \\ \Longrightarrow & \Sigma, \mathsf{let}\ x = \underline{\ell} := \underline{v}\ \mathsf{in}\ K\ \downarrow \\ \equiv & \Sigma, \mathsf{let}\ x = (V := V')[V_i/x_i]\ \mathsf{in}\ K\ \downarrow\end{array}$$

Cas [ty-succ], [ty-pred] : Par induction

$$(V[V_i/x_i], [\![\Delta; \Gamma \vdash V : \text{int}]\!]\ \rho) \quad \in \quad \mathbb{R}^{\Delta'}_{\text{int}}$$

Donc il existe $m, n \in \mathbb{Z}$ tels que

$$V[V_i/x_i] = \underline{m} \quad \text{et} \quad [\![\Delta; \Gamma \vdash V : \text{int}]\!]\ \rho = m$$

Ainsi, si $\Delta'' \supseteq \Delta'$, $(K, k) \in \mathbb{R}^{\Delta''}_{x:\text{int}^{\top}}$ et $(\Sigma, S) \in \mathbb{R}^{\Delta''}_{\mathbb{S}}$ alors

$$\begin{array}{rl} & [\![\Delta; \Gamma \vdash \mathsf{succ}\ V : \mathbf{T}(\text{int})]\!]\ \rho\ k\ S = \top \\ \equiv & k\ S\ (m+1) = \top \\ \Longrightarrow & \Sigma, \mathsf{let}\ x = \mathsf{val}\ \underline{m+1}\ \mathsf{in}\ K\ \downarrow \\ \Longrightarrow & \Sigma, \mathsf{let}\ x = \underline{m} + 1\ \mathsf{in}\ K\ \downarrow \\ \equiv & \Sigma, \mathsf{let}\ x = \mathsf{succ}\ V[V_i/x_i]\ \mathsf{in}\ K\ \downarrow\end{array}$$

La même preuve est valable pour toutes les opérations élémentaires, y compris le test d'égalité.

Pour les continuations, on fait encore une induction. Le cas $K = \mathsf{val}\ x$ est évident, car $\mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ \mathsf{val}\ x$ termine toujours. On examine le cas $\mathsf{let}\ y = M\ \mathsf{in}\ K$, où $\Delta; x : A \vdash M : \mathbf{T}(B)$ et $\Delta; \vdash K : (y : B)^{\top}$. Soient $(V, d) \in \mathbb{R}^{\Delta'}_{A}$ et $(\Sigma, S) \in \mathbb{R}^{\Delta'}_{\mathbb{S}}$ tels que $[\![\Delta; \vdash \mathsf{let}\ y = M\ \mathsf{in}\ K : (x : A)^{\top}]\!]^K\ S\ d = \top$. On veut prouver que $\Sigma, \mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ (\mathsf{let}\ y = M\ \mathsf{in}\ K)\ \downarrow$. Par induction

$$(M[V/x], [\![\Delta; x : A \vdash M : \mathbf{T}(B)]\!]\ \rho) \in \mathbb{R}^{\Delta'}_{\mathbf{T}(B)}$$

et

$$(K, [\![\Delta;\vdash K : (y : B)^{\top}]\!]^K) \in \mathbb{R}^{\Delta'}_{y:B^{\top}}$$

donc

$$\begin{array}{c}[\![\Delta; x : A \vdash M : \mathbf{T}(B)]\!]\,\{x \mapsto d\}\,[\![\Delta;\vdash K : (y : B)^{\top}]\!]^K\,S \;=\; \top \\ \Longrightarrow\; \Sigma, \mathsf{let}\; y = M[V/x]\; \mathsf{in}\; K \downarrow\end{array}$$

par le lemme 3.2.3 (termes et continuations), le premier membre est

$$[\![\Delta;\vdash \mathsf{let}\; y = M\; \mathsf{in}\; K : (x : A)^{\top}]\!]^K\,S\,d$$

et le dernier implique, par [op-ret], que

$$\Sigma, \mathsf{let}\; x = \mathsf{val}\; V\; \mathsf{in}\; (\mathsf{let}\; y = M\; \mathsf{in}\; K) \downarrow$$

ce qui conclut la preuve. ■

La propriété suivante est une conséquence directe du lemme fondamental :

Proposition 3.2.7 (Adéquation)

Si $\Delta;\vdash M : \mathbf{T}(A)$, $\Delta \vdash K : (x : A)^{\top}$, $\Sigma : \Delta$ et $S \in [\![\Sigma]\!]$ alors

$[\![\Delta;\vdash M : \mathbf{T}(A)]\!]\,\{\}\,[\![\Delta;\vdash K : (x : A)^{\top}]\!]^K\; S \;=\; \top \;\Longrightarrow\; \Sigma, \mathsf{let} x = M \mathsf{in} K \downarrow$.

Correction et adéquation réunies impliquent la validité du modèle pour prouver des équivalences observationnelles :

Proposition 3.2.8

Si $\Delta;\Gamma \vdash G_i : \gamma$ pour $i = 1, 2$ et

$$[\![\Delta;\Gamma \vdash G_1 : \gamma]\!] = [\![\Delta;\Gamma \vdash G_2 : \gamma]\!]$$

alors

$$\Delta;\Gamma \vdash G_1 \approx G_2 : \gamma$$

Démonstration : Supposons que $\Sigma : \Delta$ et $C[-] : (\Delta;\Gamma \vdash \gamma) \Rightarrow (\Delta;\vdash \mathbf{T}(A))$. On a

$$\begin{array}{lll} & \Sigma, \mathsf{let}\; x = C[G_1]\; \mathsf{in}\; \mathsf{val}\; x \downarrow & \\ \Longrightarrow & \forall S \in [\![\Sigma]\!].\, [\![\Delta;\vdash C[G_1] : \mathbf{T}(A)]\!]\,\{\}\,(\lambda S.\lambda d.\top)\,S = \top & \text{Correction} \\ \Longrightarrow & \forall S \in [\![\Sigma]\!].\, [\![\Delta;\vdash C[G_2] : \mathbf{T}(A)]\!]\,\{\}\,(\lambda S.\lambda d.\top)\,S = \top & \text{Lemme 3.2.3} \\ \Longrightarrow & \Sigma, \mathsf{let}\; x = C[G_2]\; \mathsf{in}\; \mathsf{val}\; x \downarrow & \text{Adéquation} \end{array}$$

■

$$\beta T\ \frac{\Delta;\Gamma \vdash V : A_1 \qquad \Delta;\Gamma, x : A_1 \vdash M : \mathbf{T}(A_2)}{\Delta;\Gamma \vdash \mathsf{let}\ x = \mathsf{val}\ V\ \mathsf{in}\ M \approx M[V/x] : \mathbf{T}(A_2)}$$

$$\eta T\ \frac{\Delta;\Gamma \vdash M : \mathbf{T}(A)}{\Delta;\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{val}\ x \approx M : \mathbf{T}(A)}$$

$$\beta \to \frac{\Delta;\Gamma, x : A, f : A \to \mathbf{T}(B) \vdash M : \mathbf{T}(B) \qquad \Delta;\Gamma \vdash V : A}{\Delta;\Gamma \vdash (\mathsf{rec}\ f\ (x : A) = M)\ V \approx M[V/x, \mathsf{rec}\ f\ (x : A) = M/f] : \mathbf{T}(B)}$$

$$\eta \to \frac{\Delta;\Gamma \vdash V : A \to \mathbf{T}(B)}{\Delta;\Gamma \vdash (\mathsf{rec}\ f\ (x : A) = B(V\ x)) \approx V : A \to \mathbf{T}(B)}$$

$$\text{let-cc}\ \frac{\Delta;\Gamma \vdash M : \mathbf{T}(A_1) \quad \Delta;\Gamma, y : A_1 \vdash N : \mathbf{T}(A_2) \quad \Delta;\Gamma, x : A_2 \vdash P : \mathbf{T}(A_3)}{\Delta;\Gamma \vdash \mathsf{let}\ x = (\mathsf{let}\ y = M\ \mathsf{in}\ N)\ \mathsf{in}\ P \approx \mathsf{let}\ y = M\ \mathsf{in}\ (\mathsf{let}\ x = N\ \mathsf{in}\ P) : \mathbf{T}(A_3)}$$

$$\text{mono-}T\ \frac{\Delta;\Gamma \vdash \mathsf{val}\ V_1 \approx \mathsf{val}\ V_2 : \mathbf{T}(A)}{\Delta;\Gamma \vdash V_1 \approx V_2 : A}$$

Fig. 3.6 – Égalités du métalangage

3.2.4 Égalités dans le modèle, incomplétude

Dans cette section, on présente quelques équivalences validées par le modèle, et quelques contre-exemples.

3.2.5 Égalités

La première série d'équations sont celles du métalangage λ_{ml}. Parmi elles, on peut relever les règles de congruence qui dérivent de la compositionnalité de la sémantique, les règles β et η, et d'autres règles de commutation, groupées dans la figure 3.6.

Les équations qui impliquent des références qui sont valides dans le modèles incluent les propriétés de base de l'affectation et du déréférencement, ainsi qu'une règle d'échange des allocations dynamiques. Ces règles sont présentées en figure 3.7, sous forme de système d'inférence, bien qu'elles soient loin de former une axiomatisation complète des égalités du modèle. Il y a bien sûr d'autres équations, à propos des opérations arithmétiques par exemple.

Proposition 3.2.9

Chaque égalité des figures 3.6 et 3.7 est validée dans le modèle de la section 3.2.

Démonstration : On esquisse seulement la preuve de l'équation swap. Suppo-

$$\text{swap}\ \frac{\Delta;\Gamma \vdash V_1 : \sigma_1 \quad \Delta;\Gamma \vdash V_2 : \sigma_2 \quad \Delta;\Gamma, x : \sigma_1\ \mathrm{ref}, y : \sigma_2\ \mathrm{ref} \vdash N : \mathbf{T}(A)}{\Delta;\Gamma \vdash \mathsf{let}\ x = \mathsf{ref}\ V_1\ \mathsf{in}\ (\mathsf{let}\ y = \mathsf{ref}\ V_2\ \mathsf{in}\ N) \approx \mathsf{let}\ y = \mathsf{ref}\ V_2\ \mathsf{in}\ (\mathsf{let}\ x = \mathsf{ref}\ V_1\ \mathsf{in}\ N) : \mathbf{T}(A)}$$

$$\frac{\Delta;\Gamma \vdash V_1 : \sigma\ \mathrm{ref} \quad \Delta;\Gamma \vdash V_2 : \sigma}{\Delta;\Gamma \vdash \mathsf{let}\ x = V_1 := V_2\ \mathsf{in}\ !V_1 \approx \mathsf{let}\ x = V_1 := V_2\ \mathsf{in}\ \mathsf{val}\ V_2 : \mathbf{T}(\sigma)}$$

$$\frac{\Delta;\Gamma \vdash V_1 : \sigma\ \mathrm{ref} \quad \Delta;\Gamma \vdash V_2 : \sigma \quad \Delta;\Gamma \vdash V_3 : \sigma}{\Delta;\Gamma \vdash (V_1 := V_2\ ;\ V_1 := V_3) \approx (V_1 := V_3) : \mathbf{T}(\mathsf{U})}$$

$$\frac{\Delta;\Gamma \vdash V : \sigma\ \mathrm{ref} \quad \Delta;\Gamma, x : \sigma, y : \sigma \vdash N : \mathbf{T}(A)}{\Delta;\Gamma \vdash \mathsf{let}\ x = !V\ \mathsf{in}\ (\mathsf{let}\ y = !V\ \mathsf{in}\ N) \approx \mathsf{let}\ x = !V\ \mathsf{in}\ N[x/y] : \mathbf{T}(A)}$$

Fig. 3.7 – Quelques équations à propos de références

sons que

$$\begin{array}{l}\Delta;\Gamma \vdash V_1 : \sigma_1\\ \Delta;\Gamma \vdash V_2 : \sigma_2\\ \Delta;\Gamma, x : \sigma_1\ \mathrm{ref}, y : \sigma_2\ \mathrm{ref} \vdash N : \mathbf{T}(A)\end{array}$$

et $\rho \in [\![\Gamma]\!]$, $S : \Delta$, $k : \mathbb{S} \Rightarrow [\![A]\!] \Rightarrow \mathbb{O}$. On choisit alors des adresses distinctes ℓ_1, ℓ_2 telles que

$$\begin{array}{rcl}\ell_1, \ell_2 & \notin & supp(\rho) \cup supp(k) \cup supp(S) \cup [\![\Delta;\Gamma \vdash V_1 : \sigma_1]\!]\\ & & \cup [\![\Delta;\Gamma \vdash V_2 : \sigma_2]\!] \cup [\![\Delta;\Gamma, x : \sigma_1\ \mathrm{ref}, y : \sigma_2\ \mathrm{ref} \vdash N : \mathbf{T}(A)]\!]\end{array}$$

Un calcul facile donne

$$\begin{array}{rcl} & & [\![\Delta;\Gamma \vdash \mathsf{let}\ x = \mathsf{ref}\ V_1\ \mathsf{in}\ (\mathsf{let}\ y = \mathsf{ref}\ V_2\ \mathsf{in}\ N)]\!]\,\rho k\, S\\ & = & [\![\Delta;\Gamma \vdash N : \mathbf{T}(A)]\!]\,\rho[x \mapsto \ell_1, y \mapsto \ell_2]\,k\,S'\end{array}$$

où

$$\begin{array}{rcl}S' & = & S[\ell_1 \mapsto in_{\sigma_1}([\![\Delta;\Gamma \vdash V_1 : \sigma_1]\!]\,\rho), \ell_2 \mapsto in_{\sigma_2}([\![\Delta;\Gamma \vdash V_2 : \sigma_2]\!]\,\rho[x \mapsto \ell_1])]\\ & = & S[\ell_1 \mapsto in_{\sigma_1}([\![\Delta;\Gamma \vdash V_1 : \sigma_1]\!]\,\rho), \ell_2 \mapsto in_{\sigma_2}([\![\Delta;\Gamma \vdash V_2 : \sigma_2]\!]\,\rho)]\end{array}$$

car x n'est pas libre dans V_2. On a choisi ℓ_1 et ℓ_2 assez nouvelles pour obtenir, par un autre calcul

$$\begin{array}{rcl} & & [\![\Delta;\Gamma \vdash \mathsf{let}\ y = \mathsf{ref}\ V_2\ \mathsf{in}\ (\mathsf{let}\ x = \mathsf{ref}\ V_1\ \mathsf{in}\ N)]\!]\,\rho k\, S\\ & = & [\![\Delta;\Gamma \vdash N : \mathbf{T}(A)]\!]\,\rho[x \mapsto \ell_1, y \mapsto \ell_2]\,k\,S''\end{array}$$

où

$$\begin{array}{rcl}S'' & = & S[\ell_2 \mapsto in_{\sigma_2}([\![\Delta;\Gamma \vdash V_2 : \sigma_2]\!]\,\rho), \ell_1 \mapsto in_{\sigma_1}([\![\Delta;\Gamma \vdash V_1 : \sigma_1]\!]\,\rho[y \mapsto \ell_2])]\\ & = & S[\ell_2 \mapsto in_{\sigma_2}([\![\Delta;\Gamma \vdash V_2 : \sigma_2]\!]\,\rho), \ell_1 \mapsto in_{\sigma_2}([\![\Delta;\Gamma \vdash V_1 : \sigma_1]\!]\,\rho)]\\ & = & S\end{array}$$

■

3.2.6 Incomplétude

On peut dégager trois raisons qui feraient que notre modèle n'est pas complétement adéquat :

- D'abord, les problèmes de séquentialité (ou parallèle) des modèles dénotationnels simples vont apparaître,
- L'utilisation de continuations permet d'exprimer des opérateurs de contrôle, alors que notre langage en est dépourvu,
- Enfin, le modèle ne représente pas précisément les espaces privés de la mémoire.

Nous développerons plus loin des outils pour essayer de corriger les deux derniers problèmes. On présente ici deux exemples de la troisième catégorie.

Meyer et Sieber [MS88] ont présenté une suite de cas pour évaluer les modèles des variables locales des langages de la famille d'Algol. Le premier d'entre eux est une sorte de ramasse-miettes (garbage collector), qu'on peut exprimer dans notre langage par la règle suivante :

$$[\mathrm{GC}]\ \frac{\Delta;\Gamma \vdash V : \sigma \quad \Delta;\Gamma \vdash N : \mathbf{T}(A)}{\Delta;\Gamma \vdash \mathsf{let}\ x = \mathsf{ref}\ V\ \mathsf{in}\ N \approx N : \mathbf{T}(A)}\ x \notin fv(N)$$

C'est une équivalence observationnelle valide, mais elle est fausse dans le modèle, car il contient des éléments pour N (ou la continuation qu'on lui donne) qui peuvent tester des conditions comme $\exists \ell \in \mathbb{L}.S(\ell) = in_{\mathbb{Z}}(3)$, ce qui rend l'initialisation visible. Ces tests ne sont évidemment pas définissables (ils examinent l'ensemble de la mémoire), mais on n'a rien fait jusqu'ici pour les éliminer.

Les règles swap et GC correspondent aux règles structurelles de congruence pour la restriction dans le π-calcul. On pourrait les voir comme un minimum pour que le modèle soit utilisé, mais elles ne sont pas validées dans beaucoup de travaux antérieurs : le modèle de mondes possibles de Levy [Lev02] n'en valide aucune, et comme le nôtre, le modèle de Stark [Sta94, Chapter 5] ne valide pas la deuxième. Ce n'est pas vraiment surprenant, car ces modèles sont proches du nôtre, même si l'utilisation des domaines FM rend la présentation beaucoup plus élémentaire que l'utilisation de catégories de foncteurs. Plotkin et Power ont examiné comment construire des monades, en particulier pour représenter les états locaux et globaux, à partir d'équations algébriques sur les opérations à décrire. Ils ont d'abord pensé [PP02b] que la règle GC était fausse dans leur modèle, mais un erratum [PP02a] prouve qu'elle est en fait tout à fait valide.

Le deuxième exemple de Meyer-Sieber est, dans notre syntaxe, l'équivalence suivante :

$$; f : \mathsf{U} \to \mathbf{T}(\mathsf{U}) \vdash P \approx \Omega : \mathbf{T}(\mathsf{U})$$

où P est le terme

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{ref}\ \underline{0}\ \mathsf{in}\\ f\ ();\\ \mathsf{let}\ z = !x\ \mathsf{in}\\ \mathsf{if}\ z\ \mathsf{then}\ \Omega\ \mathsf{else}\ \mathsf{val}\ ()\ \mathsf{fi}\end{array}$$

Il n'est pas difficile de se convaincre que cette équivalence est valide : f n'a pas accès à la nouvelle variable x, donc soit $f()$ diverge, soit il termine et x contient encore 0, donc le test diverge. Malheureusement, les dénotations des deux termes ne sont pas égales dans le modèle.

La dénotation de P dans un environnement ρ est

$$\lambda k.\lambda S.\,\rho(f)\ *\ \left(\lambda S'\alpha.\begin{cases} kS'* & \text{si } S'\ell = \mathsf{in}_{\mathbb{Z}} n \text{ pour un } n \neq 0\\ \bot & \text{otherwise}\end{cases}\right) S[\ell \to 0]$$

pour une adresse ℓ bien choisie. La fonction

$$\varphi = \lambda x.\lambda k.\lambda S.k\,(\lambda \ell.in_{\mathbb{Z}}1)\,*$$

qui remplit la mémoire avec des 1, n'est pas définissable, mais est un élément de $[\![\mathsf{U} \to \mathbf{T}(\mathsf{U})]\!]$. Dans un environnement où $\rho(f) = \varphi$, on obtient pour $[\![P]\!]$

$$\lambda k.\lambda S.k\,(\lambda \ell.in_{\mathbb{Z}}1)\,*$$

qui n'est manifestement pas $\bot$.

3.3 Relations paramétriques

Pour résoudre les problèmes vus dans la section précédente, on instrumente le modèle avec des relations logiques. On utilise une variante plus complexe : les relations paramétriques. Beaucoup d'auteurs ont utilisé la paramétricité à propos de la mémoire. Notre approche est très influencée par les travaux de Pitts et Stark [PS98] et de Reddy et Yang [RY04]. Il s'agit en fait de définir une famille de relations pour chaque "état" de la mémoire : les paramètres représentent les états sur lesquels s'appliqueront les éléments à relier. On utilise un ensemble ordonné : les paramètres plus grands représentent des états sur lesquels on sait plus de choses, mais les informations supplémentaires doivent rester compatibles avec les contraintes qu'on avait sur le plus petit paramètre.

Plus concrètement, on va dans un premier temps définir un ensemble ordonné $(\mathcal{P}, \rhd)$ de paramètres, puis une famille de relations sur les états indexée par les paramètres

$$\forall p \in \mathcal{P}.\,\mathcal{R}_{\mathbb{S}}(p) \subseteq \mathbb{S}_{\bot} \times \mathbb{S}_{\bot}$$

et enfin une famille de relations indexée par les paramètres et les types sur les FM-cpo utilisés pour interpréter les types :

$$\forall p \in \mathcal{P}. \forall \gamma. \mathcal{R}_\gamma(p) \subseteq [\![\gamma]\!] \times [\![\gamma]\!].$$

On va alors prouver que l'interprétation de chaque règle de typage préserve les relations, et donc que les dénotations des termes sont reliées à elles-mêmes. On en déduira que les termes qui ont des dénotations reliées sont observationnellement équivalents.

3.3.1 Relations finitaires entre états et paramètres

Les briques de base de notre système de relations sont les relations entre états. Leur particularité est qu'elles ne dépendent que d'une partie finie de la mémoire, ce qui va nous permettre de modifier les états hors de ce "domaine" en étant sûrs de préserver la relation.

La logique de la séparation (separation logic, [ORY01]) utilise cette notion de "domaine" d'une relation, mais elle est implicite : la conjonction $*$ est définie entre états partiels à supports disjoints. Nous avons choisi d'utiliser des états totaux, il va donc falloir expliciter quelle partie de ces états est concernée par la relation : la conjonction reliera en fait des états totaux tels que ces parties sont disjointes.

On pourrait penser que la notion de support des FM-cpo serait utile pour définir ce "domaine" : une idée naturelle serait de définir les relations comme des fonctions à support fini de $\mathbb{S} \times \mathbb{S} \to 1 + 1$. Malheureusement, il apparaît que ce n'est pas le bon outil pour définir la séparation des relations. D'abord, il faut "suivre les pointeurs" : par exemple, la relation

$$\{S_1, S_2 \mid \exists \ell, \ell_1, \ell_2. S_1(\ell_1) = S_2(\ell_2) = in_{\mathbb{Z}} 0 \wedge S_1(\ell) = in_{\mathbb{L}} \ell_1 \wedge S_2(\ell) = in_{\mathbb{L}} \ell_2\}$$

n'a que ℓ dans son support, mais une écriture dans l'adresse "du milieu" (quantifiée existentiellement, donc hors du support) peut briser la relation entre deux états.

Même avec une mémoire "plate", c.-à-d. où les seules valeurs enregistrables sont les entiers, une relation comme $\{(S_1, S_2) \mid \exists \ell, S_1 \ell = 0 = S_2 \ell\}$ peut être perturbée par une écriture hors de son support (qui est vide) : le but du support était d'éviter les fonctions à domaine infini, mais cet exemple montre qu'une façon facile d'avoir un support fini (vide) est précisément d'avoir un comportement infinitaire tant que toutes les adresses sont traitées uniformément. On pourrait éviter ces cas en imposant qu'une relation ne doit dépendre que des adresses accessibles en suivant des pointeurs depuis son support, mais il faudrait alors prouver que toutes les relations que nous allons

construire ont cette propriété, ce qui se révèle assez vite pénible, et interdit par exemple de modifier ℓ' dans des états reliés par $\{(S_1, S_2) \mid \exists \ell', S_1\ell = \ell' = S_2\ell\}$: la relation ne lit pas ℓ' (on peut donc y écrire sans danger), mais cette adresse est accessible depuis le support.

Nous avons plutôt opté pour une définition explicite de support, que nous appelons fonction d'accessibilité. La technique des FM-cpo est donc presque totalement absente de cette partie, mais il reste néanmoins très utile d'avoir une notion de renommage intégrée aux domaines, car la création de noms est simplifiée : au lieu d'une bijection entre noms (comme dans [RY04]), nous pouvons utiliser l'identité, c'est-à-dire représenter les noms publics par un sous-ensemble fini de $\mathbb{L}$, en choisissant la même adresse des deux côtés dans la preuve du lemme 3.3.16.

Nous commençons donc par définir les fonctions d'accessibilité : elles représentent la partie de la mémoire qu'une relation entre états inspecte. Comme cette partie dépend de l'état lui-même (c'est intuitivement l'ensemble des adresses accessibles en suivant les pointeurs depuis un ensemble d'adresses de base), on utilise des fonctions des états dans les parties finies de $\mathbb{L}$. On aurait pu les définir simplement comme la clôture par déréférencement d'un ensemble de base (le "support" de la relation), mais, pour éviter des complications techniques, nous avons préféré une définition plus abstraite, où on requiert seulement une propriété qui suffit à prouver le lemme fondamental.

Définition 3.3.1 (Fonction d'accessibilité)

Une fonction d'accessibilité A est une fonction de $\mathbb{S}$ dans les ensembles finis de $\mathbb{L}$ telle que :

$$\forall S, S' \in \mathbb{S}, (\forall \ell \in AS, S\ell = S'\ell) \Longrightarrow A(S) = A(S')$$

L'ordre de sous-typage $<:$ est défini point par point :

$$A <: A' \iff \forall S, A(S) \supseteq A'(S)$$

Une fonction d'accessibilité est plus petite qu'une autre si les contraintes induites sont plus fortes, i.e. l'ensemble des adresses examinées est plus grand. C'est un ordre partiel et la fonction $\lambda S.\emptyset$, qu'on note $\emptyset$, est le plus grand élément.

Une source de fonctions d'accessibilité est notre notion de type d'état.

Définition 3.3.2 (Partie accessible d'un état typé)

Si Δ est un type d'état, alors $\mathrm{Acc}_\Delta : \mathbb{S} \to \mathbb{P}_{fin}(\mathbb{L})$ est défini par

$$\mathrm{Acc}_\Delta(S) = \bigcup_{(\ell:\sigma)\in\Delta} \mathrm{Acc}(\ell, \sigma, S)$$

où

$$\begin{array}{rcl} \mathrm{Acc}(\ell, \mathsf{N}, S) & = & \{\ell\} \qquad \text{et} \\ \mathrm{Acc}(\ell, \sigma\ \mathrm{ref}, S) & = & \{\ell\} \cup \begin{cases} \mathrm{Acc}(\ell', \sigma, S) & \text{si } S\,\ell = \mathsf{in}_{\mathbb{L}}\ell' \\ \emptyset & \text{sinon} \end{cases} \end{array}$$

Il faut noter que pour un état S de type Δ, la clause "sinon" est exclue.

Lemme 3.3.3

1. Acc_Δ est une fonction d'accessibilité.
2. Si $\Delta \subseteq \Delta'$ alors $\mathrm{Acc}_{\Delta'} <: \mathrm{Acc}_\Delta$.

On définit maintenant ce qu'on entend par équivalence entre états selon une fonction d'accessibilité.

Définition 3.3.4

Si A est une fonction d'accessibilité, on définit

$$S \sim S' : A \iff \forall \ell \in A(S), S\ell = S'\ell$$

Il est aisé de vérifier que $\cdot \sim \cdot : A$ définit bien une relation d'équivalence.

Définition 3.3.5 (Relation finitaire entre états)

Une relation finitaire entre états r est une paire $\langle |r|, A_r \rangle$ où $|r| \subseteq \mathbb{S} \times \mathbb{S}$ et A_r est une fonction d'accessibilité, vérifiant la condition de saturation suivante : si $S_1 \sim S_1' : A_r$ and $S_2 \sim S_2' : A_r$ alors

$$(S_1, S_2) \in |r| \iff (S_1', S_2') \in |r|$$

On définit la relation correspondant à l'équivalence entre états à un type Δ :

Définition 3.3.6 (Relation identité)

Si Δ est un type d'état, on définit $id_\Delta = \langle |id_\Delta|, \mathrm{Acc}_\Delta \rangle$, où

$$(S_1, S_2) \in |id_\Delta| \iff S_1 \sim S_2 : \Delta$$

Lemme 3.3.7

1. $id_\Delta = \langle |id_\Delta|, \mathrm{Acc}_\Delta \rangle$ est une relation finitaire.

2. Si A et A' sont des fonctions d'accessibilité alors $A \wedge A'$ aussi, où

$$\forall S.\, (A \wedge A')(S) = (A(S)) \cup (A'(S))$$

3. Si $S \sim S' : A$ et $A <: A'$ alors $S \sim S' : A'$.
4. $\top = \langle \mathbb{S} \times \mathbb{S}, \emptyset \rangle$ est une relation finitaire.
5. Si $\langle |r|, A \rangle$ est une relation finitaire et si $A' <: A$, alors $\langle |r|, A' \rangle$ est encore une relation finitaire.

La notion de fonction d'accessibilité permet de définir la conjonction séparante, similaire à la logique de la séparation.

Définition 3.3.8 (Conjonction séparante)

Si $r_1 = \langle |r^1|, A^1 \rangle$ et $r_2 = \langle |r^2|, A^2 \rangle$ sont des relations finitaires, on définit

$$r^1 \otimes r^2 \;=\; \langle |r^1 \otimes r^2|, A^1 \wedge A^2 \rangle$$

où $|r^1 \otimes r^2|$ est l'ensemble des $(S_1, S_2) \in |r^1| \cap |r^2| = \emptyset$ tels que $A^1(S_i) \cap A^2(S_i)$ pour $i = 1, 2$.

Lemme 3.3.9

Si r^1 et r^2 sont des relations finitaires, $r^1 \otimes r^2$ aussi. La conjonction $\otimes$ est associative et commutative (à isomorphismes près), et $\top$ est une unité.

3.3.2 Paramètres et relations paramétriques

Nous avons maintenant tous les ingrédients pour définir les paramètres de nos relations : les paramètres expriment qu'une partie de la mémoire est directement accessible par le contexte (elle est dite visible), et qu'une autre partie est uniquement connue du programme. La partie visible pouvant être modifiée librement par le contexte, on sait seulement que les états resteront égaux sur ces adresses. Au contraire, la seule façon de modifier la partie privée de la mémoire est d'utiliser le terme, il est donc possible d'énoncer des invariants sur cette partie. Nos paramètres seront donc formés de deux parties : un ensemble fini d'adresses visibles, et un invariant sur la partie cachée.[2]

[2]Nous avons en cela des paramètres similaires à ceux de Reddy et Yang [RY04]. Pitts et Stark [PS98] utilisent une mémoire plate, ou l'ensemble des adresses visibles est implicite : on peut rendre une adresse ℓ visible dans le paramètre r en utilisant le paramètre $r \otimes id_{\{\ell\}}$. En présence de références sur les références, cette définition ne convient plus car elle interdit aux adresses visibles de pointer les unes vers les autres.

Définition 3.3.10 (Paramètres)

Un paramètre est une paire (Δ, r), où Δ est un type d'état et r est une relation finitaire entre états. On écrit simplement Δr, et $\mathcal{P}$ pour l'ensemble des paramètres.

Pour les valeurs :

$$\begin{array}{rcl}
\mathcal{R}_{\mathsf{U}}(\Delta r) & = & \{(*,*)\} \\
\mathcal{R}_{\mathsf{N}}(\Delta r) & = & \{(n,n) \mid n \in N\} \\
\mathcal{R}_{\sigma\ \mathrm{ref}}(\Delta r) & = & \{(\ell,\ell) \mid (\ell:\sigma) \in \Delta\} \\
\mathcal{R}_{A\to\mathbf{T}(A')}(\Delta r) & = & \{(f_1,f_2) \mid \forall \Delta' r' \rhd \Delta r, \forall (v_1,v_2) \in \mathcal{R}_A(\Delta' r'), \\
 & & \qquad (f_1 v_1, f_2, v_2) \in \mathcal{R}_{\mathbf{T}(A')}(\Delta' r')\}
\end{array}$$

Pour les continuations, $(k_1,k_2) \in \mathcal{R}_{A^\top}(\Delta r)$ si et seulement si

$$\forall \Delta' r' \rhd \Delta r, (v_1,v_2) \in \mathcal{R}_A(\Delta' r'), (S_1,S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r'), k_1 S_1 v_1 = k_2 S_2 v_2$$

Et pour les programmes $(f_1,f_2) \in \mathcal{R}_{\mathbf{T}(A)}(\Delta r)$ si et seulement si

$$\forall \Delta' r' \rhd \Delta r, (k_1,k_2) \in \mathcal{R}_{A^\top}(\Delta' r'), (S_1,S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r'), f_1 k_1 S_1 = f_2 k_2 S_2$$

Fig. 3.8 – Définition des relations paramétriques

Si Δr est un paramètre, on définit une relation finitaire

$$\mathcal{R}_{\mathbb{S}}(\Delta r) \ = \ |id_\Delta \otimes r|$$

$\mathcal{P}$ est ordonné par $\rhd$:

$$\Delta r \rhd \Delta' r' \iff (\Delta \supseteq \Delta') \wedge (\exists r'', r = r' \otimes r'')$$

Les relations $\mathcal{R}_\gamma(\Delta r)$ proprement dites sont très semblables à celles de Pitts et Stark, car notre langage est voisin du leur. Elles sont définies par induction sur les types, comme indiqué en figure 3.8.

On aurait pu définir $\mathcal{R}_{\sigma\ \mathrm{ref}}(\cdot)$ comme "les couples d'adresses telles que l'écriture, la lecture et le test d'égalité sont des programmes reliés", comme font Pitts et Stark [PS98]. Bien que cette définition permette de prouver le lemme fondamental, et donc de valider le raisonnement par relations, elle introduit beaucoup de complications[3]. Comme nous avons ici un ensemble Δ d'adresses visibles (au contraire de [PS98]), nous avons préféré l'utiliser pour définir $\mathcal{R}_{\sigma\ \mathrm{ref}}(\Delta r)$, comme le font Reddy et Yang [RY04].

[3]En fait, il faudrait prouver que toutes les relations que nous considérons plus loin se comportent comme attendu, c.-à-d. que les adresses visibles sont reliées à elles-mêmes. C'est vrai, mais la preuve est assez acrobatique, et ne marcherait pas si les référence sur U étaient autorisées (on ne peut pas observer une affectation dans ces références), et elle repose sur le fait que les relations sont non vides (comme dans [PS98]), ce qui complique les choses puisque le produit $\otimes$ est alors partiel.

Le lemme suivant est très utile :

Lemme 3.3.11 (Affaiblissement)

Pour toute relation paramétrique

$$\Delta' r' \rhd \Delta r \implies \begin{cases} \mathcal{R}_A(\Delta' r') \supseteq \mathcal{R}_A(\Delta r) \\ \mathcal{R}_{A^\top}(\Delta' r') \supseteq \mathcal{R}_{A^\top}(\Delta r) \\ \mathcal{R}_{\mathbf{T}(A)}(\Delta' r') \supseteq \mathcal{R}_{\mathbf{T}(A)}(\Delta r) \end{cases}$$

Démonstration : Facile. ■

Les relations sur les valeurs et et les programmes coïncident dans le sens suivant :

Lemme 3.3.12 (Coïncidence)

Pour tous les types A, si $v_1, v_2 \in [\![A]\!]$:

$$(v_1, v_2) \in \mathcal{R}_A(\Delta r) \Longrightarrow (\mathsf{val}\, v_1, \mathsf{val}\, v_2) \in \mathcal{R}_{\mathbf{T}(A)}(\Delta r)$$

où $\mathsf{val}\, v = \lambda k.\lambda S.kSv$.

Démonstration : Supposons que $(v_1, v_2) \in \mathcal{R}_A(\Delta r)$. Soient $\Delta' r' \rhd \Delta r$, $(k_1, k_2) \in \mathcal{R}_{A^\top}(\Delta' r')$, $(s_1, s_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r')$. Par le lemme d'affaiblissement, $(v_1, v_2) \in \mathcal{R}_A(\Delta' r')$, donc

$$(\mathsf{val}\, v_1)k_1 s_1 = k_1 s_1 v_1 = k_2 s_2 v_2 = (\mathsf{val}\, v_2)k_2 s_2$$

■

La réciproque est vraie pour les relations non triviales :

Lemme 3.3.13

Si $\mathcal{R}_{\mathbb{S}}(\Delta r)$ est non vide, pour tout $v_1, v_2 \in [\![A]\!]$

$$(\mathsf{val}\, v_1, \mathsf{val}\, v_2) \in \mathcal{R}_{\mathbf{T}(A)}(\Delta r) \Longrightarrow (v_1, v_2) \in \mathcal{R}_A(\Delta r)$$

Démonstration : Cas U : Le seul couple (v_1, v_2) possible est $(*, *)$, qui est dans $\mathcal{R}_{\mathsf{U}}(\Delta r)$.

Cas N : Supposons que $(\mathsf{val}\, n, \mathsf{val}\, m) \in \mathcal{R}_{\mathbf{T}(\mathsf{N})}(\Delta r)$. On définit

$$kSn' = \begin{cases} \top & \text{si } n' = n \\ \bot & \text{sinon} \end{cases}$$

Il est facile de vérifier que $(k, k) \in \mathcal{R}_{\mathsf{N}^\top}(\Delta r)$. Comme $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(r)$,

$$\top = kS_1 n = (\mathsf{val}\, n)kS_1 = (\mathsf{val}\, m)kS_2 = kS_2 m$$

on obtient $m = n$.

Cas σ ref : Supposons $(\mathsf{val}\,\ell_1, \mathsf{val}\,\ell_2) \in \mathcal{R}_{\mathbf{T}(\sigma\ \mathrm{ref})}(\Delta r)$. Comme pour les entiers, on peut prouver que $\ell_1 = \ell_2$. Il suffit donc de vérifier que $\Delta(\ell_1) = \sigma$. Supposons que c'est faux, et définissons

$$k_1 = \lambda S.\lambda \ell.\text{si } \ell = \ell_1 \text{ alors } \top \text{ sinon } \bot$$
$$k_2 = \lambda S.\lambda \ell.\bot$$

Comme $\Delta\ell_1 \neq \sigma$, $(k_1, k_2) \in \mathcal{R}_{\sigma\ \mathrm{ref}^\top}(\Delta r)$, mais comme il existe $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta r)$, $(\mathsf{val}\,\ell_1)k_1S_1 \neq (\mathsf{val}\,\ell_2)k_2S_2$, on obtient une contradiction.

Cas $[A_1 \to TA_2]$: Supposons que $(\mathsf{val}\,f_1, \mathsf{val}\,f_2) \in \mathcal{R}_{\mathbf{T}(A_1 \to \mathbf{T}(A_2))}(\Delta r)$, Soient $\Delta^1 r^1 \triangleright \Delta r$ et $(v_1^1, v_2^1) \in \mathcal{R}_{A_1}(\Delta^1 r^1)$. On veut prouver que $(f_1 v_1^1, f_2 v_2^1) \in \mathcal{R}_{\mathbf{T}(A_2)}(\Delta^1 r^1)$. Soient $\Delta^2 r^2 \triangleright \Delta^1 r^1$, $(k_1^2, k_2^2) \in \mathcal{R}_{A_2{}^\top}(\Delta^2 r^2)$ et $(s_1^2, s_2^2) \in \mathcal{R}_{\mathbb{S}}(\Delta^2 r^2)$, on définit alors

$$k_i' = \lambda S.\lambda f.f v_i^1 k_i^2 S$$

D'abord, prouvons que $(k_1', k_2') \in \mathcal{R}_{A_1 \to \mathbf{T}(A_2)^\top}(\Delta^2 r^2)$. Supposons que $\Delta^3 r^3 \triangleright \Delta^2 r^2$, $(f_1^3, f_2^3) \in \mathcal{R}_{A_1 \to \mathbf{T}(A_2)}(\Delta^3 r^3)$ et $(s_1^3, s_2^3) \in \mathcal{R}_{\mathbb{S}}(\Delta^3 r^3)$. Par affaiblissement, $(k_1^2, k_2^2) \in \mathcal{R}_{A_2{}^\top}(\Delta^3 r^3)$ et $(v_1^1, v_2^1) \in \mathcal{R}_{A_1}(\Delta^3 r^3)$, ce qui nous donne :

$$k_1' f_1^3 s_1^3 = f_i^3 v_i^1 k_1^2 s_1^3 = f_2^3 v_2^1 k_2^2 s_2^3 = k_2' f_2^3 s_2^3$$

On peut alors conclure

$$(f_1 v_1) k_1 S_1 = k_1' S_1 f_1 = (\mathsf{val}\ f_1) k_1' S_1 = (\mathsf{val}\ f_2) k_2' S_2 = k_2' S_2 f_2 = (f_2 v_2) k_2 S_2$$

■

Le lemme suivant prouve qu'on peut modifier les états tout en les gardant reliés par $\mathcal{R}_{\mathbb{S}}(\Delta r)$, soit en écrivant deux valeurs reliées du bon type à la même adresse visible, soit en écrivant à une nouvelle adresse et en l'ajoutant aux adresses visibles avec le bon type.

Lemme 3.3.14

Supposons $(v_1, v_2) \in \mathcal{R}_\sigma(\Delta r)$ et $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta r)$. Si $\Delta(\ell) = \sigma$, ou si ℓ est une nouvelle adresse, c'est-à-dire que, pour $i = 1, 2$, $\ell \notin A_{id_\Delta(S_i) \otimes r}(S)$, alors

$$(S_1[\ell \to v_1], S_2[\ell \to v_2]) \in \mathcal{R}_{\mathbb{S}}(\Delta[\ell \to \sigma] r)$$

Démonstration : On note $\Delta' = \Delta[\ell \to \sigma]$. et $S_i' = S_i[\ell \to v]$. D'abord, il faut remarquer que $v_1 = v_2$, qu'on note v. Il faut prouver que $S_1' \sim S_2' : \Delta'$, $\mathrm{Acc}_{\Delta'}(S_i') \cap A_r(S_i') = \emptyset$, et $(S_1', S_2') \in |r|$.

Si ℓ est une nouvelle adresse, alors $\ell \notin \mathrm{Acc}_\Delta(S_1) \cup \mathrm{Acc}_\Delta(S_2)$, donc $S'_1 \sim S'_2 : \Delta$. Comme $(v_1, v_2) \in \mathcal{R}_\sigma(\Delta r)$, $S'_1 \sim S'_2 : (\ell : \sigma)$, ce qui nous donne $S'_1 \sim S'_2 : \Delta[\ell \to \sigma]$. Si $\Delta\ell = \sigma$, on prouve par induction sur σ' que à chaque fois que $S_1 \sim S_2 : (\ell' : \sigma')$, $S'_1 \sim S'_2 : (\ell' : \sigma')$ également :

- N Si $\ell' = \ell$, alors $\sigma' = \sigma$. On a $S'_1\ell' = v = S'_2\ell'$ et $v \in \mathbb{Z}$, donc $S'_1 \sim S'_2 : (\ell' : \sigma')$. Si $\ell' \neq \ell$, alors $S'_1\ell' = S_1\ell' = n = S_2\ell' = S'_2\ell'$ pour un entier n, donc $S'_1 \sim S'_2 : (\ell' : \sigma')$ aussi.
- σ'' ref On considère encore deux cas. Si $\ell' = \ell$, alors $\sigma' = \sigma$, donc $v = \ell''$ pour une adresse $\ell'' \in \mathbb{L}$ telle que $\Delta\ell'' = \sigma''$. Comme $(S_1, S_2) \in id_\Delta$, $S_1 \sim S_2 : (\ell'' : \sigma'')$, l'hypothèse d'induction nous donne $S'_1 \sim S'_2 : (\ell'' : \sigma'')$, ce qui implique que $S'_1 \sim S'_2 : (\ell' : \sigma')$. Si $\ell' \neq \ell$, alors $S'_i\ell' = S_i\ell'$. Comme $(S_1, S_2) \in id_\Delta$, il existe ℓ'' tels que $S_1\ell' = \ell'' = S_2\ell'$ et $S_1 \sim S_2 : (\ell'' : \sigma'')$. Encore par hypothèse d'induction, $S'_1 \sim S'_2 : (\ell'' : \sigma'')$, ce qui implique que $S'_1 \sim S'_2 : (\ell' : \sigma')$.

Maintenant, prouvons que $\mathrm{Acc}_\Delta(S'_i) \cap A_r(S'_i) = \emptyset$. D'abord, comme la seule adresse modifiée est $\ell \notin A_r(S_1) \cup A_r(S_2)$,

$$A_r(S'_i) = A_r(S_i)$$

On prouve facilement que $\mathrm{Acc}_{\Delta'}(S'_i) \subseteq \mathrm{Acc}_\Delta(S_i) \cup \{\ell\}$, ce qui donne

$$\mathrm{Acc}_{\Delta'}(S'_i) \cap A_r(S'_i) \subseteq \mathrm{Acc}_\Delta(S_i) \cap A_r(S_i) = \emptyset$$

Comme $A_r(S_i)$ reste inchangé et que $(S_1, S_2) \in |r|$, $(S'_1, S'_2) \in |r|$, ce qui conclut la preuve. ■

Lemme 3.3.15

Si $(v_1, v_2) \in \mathcal{R}_{\sigma\ \mathrm{ref}}(\Delta r)$ et $(v'_1, v'_2) \in \mathcal{R}_\sigma(\Delta r)$, alors $(v_1 := v'_1, v_2 := v'_2) \in \mathcal{R}_{\mathbf{T}(\mathsf{U})}(\Delta r)$.

Démonstration : C'est une conséquence immédiate du lemme 3.3.14, dans le cas $\Delta\ell = \sigma$. ■

Lemme 3.3.16

Si $(v_1, v_2) \in \mathcal{R}_\sigma(\Delta r)$, alors $(\mathsf{ref}\ v_1, \mathsf{ref}\ v_2) \in \mathcal{R}_{\mathbf{T}(\sigma\ \mathrm{ref})}(\Delta r)$.

Démonstration : Soient $\Delta' r' \triangleright \Delta r$, $(k_1, k_2) \in \mathcal{R}_{\sigma\ \mathrm{ref}^\top}(\Delta' r')$ et $(S_1, S_2) \in \mathcal{R}_\mathbb{S}(\Delta' r')$. On choisit une nouvelle adresse ℓ, hors de $supp(v_i)$, $supp(k_i)$ et $A_{\mathcal{R}_\mathbb{S}(\Delta r)}(S_i)$ pour $i = 1, 2$. Par le lemme 3.3.14, dans le cas d'une nouvelle adresse ℓ, on obtient que $(s_1[\ell \to v_1], s_2[\ell \to v_2]) \in \mathcal{R}_\mathbb{S}(\Delta'[\ell \to \sigma] r')$. Par affaiblissement, $(k_1, k_2) \in \mathcal{R}_{\sigma\ \mathrm{ref}^\top}(\Delta'[\ell \to \sigma] r')$, et, comme $(\ell, \ell) \in \mathcal{R}_{\sigma\ \mathrm{ref}}(\Delta'[\ell \to \sigma] r')$,

$$(\mathsf{ref}\ v_1) k_1 s_1 = k_1(s_1[\ell \to v_1])\ell = k_2(s_2[\ell \to v_2])\ell = (\mathsf{ref}\ v_2) k_2 s_2$$

■

Les autres opérations sur la mémoire préservent elles aussi les relations.

Lemme 3.3.17

Pour tout Δr, $(v_1, v_2) \in \mathcal{R}_{\Gamma\vdash\sigma\ \mathsf{ref}}(\Delta r) \implies (!v_1, !v_2) \in \mathcal{R}_{\Gamma\vdash\sigma}(\Delta r)$

Démonstration : Soient $\Delta' r' \rhd \Delta r$, et $(k_1, k_2) \in \mathcal{R}_{\sigma\ \mathsf{ref}^\top}(\Delta' r')$. Il existe une adresse ℓ telle que $v_1 = \ell = v_2$ et $\Delta\ell = \sigma$. Comme $\Delta' \supseteq \Delta$, $\Delta'\ell = \sigma$ aussi.

Si $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r')$, alors, comme $|id_{\Delta'} \otimes r'| \subseteq |id_{\Delta'}| \cap |r'|$, $S_1 \sim S_2 : \Delta'$. En particulier $S_1 \sim S_2 : (\ell : \sigma)$: il existe v' tels que $S_1\ell = \mathsf{in}_{[\![\sigma]\!]}v' = S_2\ell$, et

$$!v_i k_i S_i = k_i S_i v'$$

Si $\sigma = \mathsf{N}$, alors $v' \in \mathbb{Z}$, et donc $(v', v') \in \mathcal{R}_\sigma(\Delta' r')$, ce qui permet de conclure. Si $\sigma = \sigma'\ \mathsf{ref}$, alors $v' \in \mathbb{L}$. Si $v' \in dom\Delta$, alors v' a le type σ', car S_1 et S_2 ont le type Δ. On obtient que $(v', v') \in \mathcal{R}_\sigma(\Delta' r')$. Si $v' \notin dom\Delta$, soit $\Delta'' = \Delta' \cup \{v' : \sigma'\}$. Comme $\Delta'' r' \rhd \Delta' r'$, par affaiblissement, $(k_1, k_2) \in \mathcal{R}_{\sigma\ \mathsf{ref}^\top}(\Delta'' r')$ et $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta'' r')$. Puisque $(v', v') \in \mathcal{R}_\sigma(\Delta'' r')$, on a bien $k_1 S_1 v' = k_2 S_2 v'$. ■

Lemme 3.3.18

Pour tout Δr,

$$\begin{cases} (v_1, v_2) \in \mathcal{R}_{\Gamma\vdash\sigma\ \mathsf{ref}}(\Delta r) \\ (v_1', v_2') \in \mathcal{R}_{\Gamma\vdash\sigma\ \mathsf{ref}}(\Delta r) \end{cases} \implies (v_1 = v_1', v_2 = v_2') \in \mathcal{R}_{\Gamma\vdash\mathbf{T}(\mathsf{N})}(\Delta r)$$

Démonstration : Par définition $v_1 = v_2$ et $v_1' = v_2'$, donc soit $v_1 = v_1'$ et $v_2 = v_2'$, soit $v_1 \neq v_1'$ et $v_2 \neq v_2'$. Dans les deux cas, les résultats sont reliés. ■

Pour prouver que les règles de typage préservent les relations, il faut considérer des dénotations ouvertes.

Définition 3.3.19 (Relations sous contexte)

Si Γ est un contexte, on définit $(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta r)$ comme

$$\forall (x_i : A_i) \in \Gamma, (\rho_1 x_i, \rho_2 x_i) \in \mathcal{R}_{A_i}(\Delta r)$$

et $(v_1, v_2) \in \mathcal{R}_{\Gamma\vdash\gamma}(\Delta r)$ comme

$$\forall \Delta' r' \rhd \Delta r, \forall (\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta' r'), (v_1\rho_1, v_2\rho_2) \in \mathcal{R}_\gamma(\Delta' r')$$

Lemme 3.3.20

Les dénotations des règles de typage préservent les relations. Plus précisément, pour tout Δr

[ty-unit] $(*,*) \in \mathcal{R}_{\Gamma\vdash \mathsf{N}}(\Delta r)$,
[ty-int] $(n,n) \in \mathcal{R}_{\Gamma\vdash \mathsf{N}}(\Delta r)$,
[ty-var] $([\![\Delta;\Gamma,x:A\vdash x:A]\!],[\![\Delta;\Gamma,x:A\vdash x:A]\!]) \in \mathcal{R}_{\Gamma,x:A\vdash A}(\Delta r)$,
[ty-rec] Si $(f_1,f_2) \in \mathcal{R}_{\Gamma,x:A,f:A\to \mathbf{T}(A')\vdash \mathbf{T}(A')}(\Delta r)$ alors $(\lambda\rho.fix(F_1\rho),\lambda\rho.fix(F_2\rho)) \in \mathcal{R}_{\Gamma\vdash A\to\mathbf{T}(A')}(\Delta r)$, où $F_i\rho = \lambda f'.\lambda x'.f_i\rho[x\to x', f\to f']$,
[ty-val] Si $(v_1,v_2)\in\mathcal{R}_{\Gamma\vdash A}(\Delta r)$, alors $(\mathsf{val}\ v_1, \mathsf{val}\ v_2)\in\mathcal{R}_{\Gamma\vdash\mathbf{T}(A)}(\Delta r)$,
[ty-let] Si $(m_1,m_2)\in\mathcal{R}_{\Gamma\vdash\mathbf{T}(A)}(\Delta r)$ et $(m_1',m_2')\in\mathcal{R}_{\Gamma,x:A\vdash\mathbf{T}(A)'}(\Delta r)$ alors

$$(\mathsf{let}\ x{=}m_1\ \mathsf{in}\ m_1', \mathsf{let}\ x{=}m_2\ \mathsf{in}\ m_2') \in \mathcal{R}_{\Gamma\vdash\mathbf{T}(A)'}(\Delta r)$$

où $\mathsf{let}\ x{=}m\ \mathsf{in}\ m' = \lambda\rho.\lambda k.\lambda S.m\rho(\lambda S'.\lambda d.m'\rho[x\to d]kS')S$
[ty-app] Si $(v_1,v_2)\in\mathcal{R}_{\Gamma\vdash A\to\mathbf{T}(A)'}(\Delta r)$ et $(v_1',v_2')\in\mathcal{R}_{\Gamma\vdash A}(\Delta r)$, alors $(\lambda\rho.v_1\rho(v_1'\rho),\lambda\rho,v_2\rho(v_2'\rho))\in\mathcal{R}_{\Gamma\vdash\mathbf{T}(A)'}(\Delta r)$,
[ty-if] Si $(v_1,v_2)\in\mathcal{R}_{\Gamma\vdash\mathsf{N}}(\Delta r)$, $(m_1,m_2)\in\mathcal{R}_{\Gamma\vdash\mathbf{T}(\mathsf{N})}(\Delta r)$ et $(m_1',m_2')\in\mathcal{R}_{\Gamma\vdash\mathbf{T}(\mathsf{N})}(\Delta r)$, alors

$$([\![\mathsf{if}]\!](v_1,m_1,m_1'),[\![\mathsf{if}]\!](v_2,m_2,m_2')) \in \mathcal{R}_{\Gamma\vdash\mathbf{T}(\mathsf{N})}(\Delta r)$$

ou $[\![\mathsf{if}]\!](n,m,m')$ est m si $n=0$ et m' sinon.
[ty-succ] Si $(v_1,v_2)\in\mathcal{R}_{\Gamma\vdash\mathsf{N}}(\Delta r)$, alors

$$([\![\mathsf{succ}]\!](v_1),[\![\mathsf{succ}]\!](v_2)) \in \mathcal{R}_{\Gamma\vdash\mathbf{T}(\mathsf{N})}(\Delta r)$$

[ty-pred] Si $(v_1,v_2)\in\mathcal{R}_{\Gamma\vdash\mathsf{N}}(\Delta r)$, alors

$$([\![\mathsf{succ}]\!](v_1),[\![\mathsf{succ}]\!](v_2)) \in \mathcal{R}_{\Gamma\vdash\mathbf{T}(\mathsf{N})}(\Delta r)$$

Démonstration : Les règles [ty-unit], [ty-int], [ty-var] sont des conséquences évidentes des définitions. Pour [ty-rec], on définit

$$\begin{aligned} f_i^0\rho &= \lambda x.\lambda k.\lambda s.\bot \\ f_i^{k+1}\rho &= F_i\rho(f_i^k\rho) \end{aligned}$$

Une induction nous donne que, pour tout k $(f_1^k,f_2^k)\in\mathcal{R}_{\Gamma\vdash A_1\to\mathbf{T}(A_2)}(\Delta r)$. Soit $\Delta' r' \rhd \Delta r$, $(k_1,k_2)\in\mathcal{R}_{A^\top}(\Delta' r')$ et $(S_1,S_2)\in\mathcal{R}_{\mathbb{S}}(\Delta' r')$. Comme pour tout k, $f_1^k k_1 S_1 = f_2^k k_2 S_2$, $(\sqcup f_1)k_1S_1 = \sqcup(f_1^k k_1 S_1) = \sqcup(f_2^k k_2 S_2) = (\sqcup f_2)k_2S_2 : (\sqcup f_1^k, \sqcup f_2^k)\in\mathcal{R}_{\Gamma\vdash A_1\to\mathbf{T}(A_2)}(\Delta r)$, i.e. $(fixF_1, fixF_2)\in\mathcal{R}_{\Gamma\vdash A_1\to\mathbf{T}(A_2)}(\Delta r)$.

Pour les programmes, on suppose que $\Delta'' r'' \rhd \Delta' r' \rhd r$, et que $(\rho_1,\rho_2)\in\mathcal{R}_\Gamma(\Delta' r')$, $(k_1,k_2)\in\mathcal{R}_{A_2{}^\top}(\Delta'' r'')$ et $(S_1,S_2)\in\mathcal{R}_{\mathbb{S}}(\Delta'' r'')$. Le cas [ty-val] est la direction facile du lemme de coïncidence 3.3.12. On prouve les cas intéressants restants ([ty-if], [ty-pred] et [ty-succ] sont faciles)

[ty-let] On veut :

$$m_1\rho_1(\lambda S'.\lambda d.m_1'\rho_1[x \to d]k_1S')S_1 = m_2\rho_2(\lambda S'.\lambda d.m_2'\rho_2[x \to d]k_2S')S_2$$

Soit $k_i' = \lambda S^3.\lambda d.m_i'\rho_i[x \to d]k_iS^3$. On prouve que $(k_1', k_2') \in \mathcal{R}_{A_1{}^\top}(\Delta''r'')$, et l'hypothèse sur (m_1, m_2) nous donnera alors l'égalité attendue. Supposons que $\Delta^3r^3 \rhd \Delta''r''$, $(v_1^3, v_2^3) \in \mathcal{R}_{A_1}(\Delta^3r^3)$ et $(S_1^3, S_2^3) \in \mathcal{R}_{\mathbb{S}}(\Delta^3r^3)$.

$$k_i'S_i^3v_i^3 = m_i'\rho_i[x \to v_i^3]k_iS_i^3$$

Par affaiblissement, $(\rho_1[x \to v_1^3], \rho_2[x \to v_2^3]) \in \mathcal{R}_{\Gamma,x:A_1}(\Delta^3r^3)$, donc par hypothèse sur (m_1', m_2'), on obtient $k_1'S_1^3v_1^3 = k_2'S_2^3v_2^3$.

[ty-app] $(v_1\rho_1, v_2\rho_2) \in \mathcal{R}_{A\to\mathbf{T}(A')}(\Delta'r')$ et $(v_1'\rho_1, v_2'\rho_2) \in \mathcal{R}_A(\Delta'r')$, donc $(v_1\rho_1(v_1'\rho_1'), v_2\rho_2(v_2'\rho_2')) \in \mathcal{R}_{\mathbf{T}(A')}(\Delta'r')$. ■

Nous pouvons maintenant prouver :

Théorème 3.3.21 (Lemme fondamental)

Si $\Delta;\Gamma \vdash P : \gamma$, alors $[\![\Delta;\Gamma \vdash P : \gamma]\!]$ est invariant par $\mathcal{R}_{\Gamma\vdash\gamma}(\Delta r)$ pour tout r.

Démonstration : Par induction sur P, en utilisant le lemme précédent. ■

Et finalement, on peut conclure :

Théorème 3.3.22 (Correction)

Soient P_1, P_2 deux termes tels que

$$\Delta;\Gamma \vdash P_i : A \qquad ([\![P_1]\!], [\![P_2]\!]) \in \mathcal{R}_{\Gamma\vdash\mathbf{T}(A)}(\Delta id_{\{\}})$$

Soit $C[\cdot] : (\Delta;\Gamma \vdash \mathbf{T}(A)) \Rightarrow (\Delta;- \vdash \mathbf{T}(A'))$ un contexte. Pour tout $\Sigma : \Delta$:

$$\Sigma, C[P_1]\downarrow \iff \Sigma, C[P_2]\downarrow$$

Démonstration : Soit r la relation $id_{\{\}}$. D'abord, $([\![\Sigma]\!], [\![\Sigma]\!]) \in id_\Delta$. r étant l'unité pour $\otimes$, nous avons

$$([\![\Sigma]\!], [\![\Sigma]\!]) \in \mathcal{R}_{\mathbb{S}}(\Delta r)$$

Par induction sur C, puisque les règles de typage préservent les relations :

$$([\![C[P_1]]\!], [\![C[P_2]]\!]) \in \mathcal{R}_{\mathbf{T}(A')}(\Delta r)$$

Il est évident que $\lambda sd.\top$ est invariant par $\mathcal{R}_{\mathbf{T}(A')^\top}(\Delta r)$, donc

$$[\![C[P_1]]\!](\lambda sd.\top)[\![\Sigma]\!] = [\![C[P_2]]\!](\lambda sd.\top)[\![\Sigma]\!]$$

Par correction et adéquation de la sémantique :

$$\Sigma, C[P_1]\downarrow \iff \Sigma, C[P_2]\downarrow$$

■

3.3.3 Typage des états généralisé

Les fonctions d'accessibilité sont utiles pour prouver des résultats généraux, mais s'avèrent peu pratiques à utiliser. Dans la grande majorité des exemples, il suffit de donner un type d'état à l'invariant. Dans certains cas, il faut permettre à l'invariant de pointer dans la partie visible de la mémoire (mais pas de lire dans la partie visible de la mémoire). On généralise en conséquence la notion de type d'état, en introduisant un type $\mathbb{T}$.

Définition 3.3.23 (Types enregistrables étendus)

$$\alpha ::= \mathbb{T} \mid \mathsf{N} \mid \alpha \text{ ref}$$

Le sous-typage $<:$ est défini par les règles suivantes :
- $\mathsf{N} <: \mathbb{T}$,
- $\alpha \text{ ref} <: \mathbb{T}$,
- si $\alpha <: \alpha'$, alors $\alpha \text{ ref} <: \alpha' \text{ ref}$,

Les types α disent quelle partie de la valeur stockée à une adresse est intéressante pour la relation : $\mathbb{T}$ signifie qu'on ne s'intéresse pas à cette adresse. Par exemple, une adresse a le type $\mathbb{T}$ ref si on exige qu'elle contienne une référence, mais qu'on n'inspecte pas la valeur pointée (qui pourrait donc être un entier ou une autre référence).

Définition 3.3.24 (Type d'état étendu)

Un type d'état θ est une fonction des adresses $\mathbb{L}$ dans les types enregistrables étendus, qui vaut $\mathbb{T}$ partout sauf sur un nombre fini d'adresses. Le sous typage est défini point par point.

Définition 3.3.25 (Fonction d'accessibilité pour un type étendu)

On définit $\mathrm{Acc}(\cdot : \theta)$ pour un type θ par

$$\mathrm{Acc}(s : \theta) = \bigcup_{\ell \in \mathbb{L}} \mathrm{Acc}(s \vdash \ell : \theta(\ell))$$

où
- $\mathrm{Acc}(s \vdash \ell : \mathbb{T}) = \emptyset$,
- $\mathrm{Acc}(s \vdash \ell : \mathsf{N}) = \{\ell\}$,
- $\mathrm{Acc}(s \vdash \ell : \alpha \text{ ref}) = \{\ell\} \cup \begin{cases} \mathrm{Acc}(s \vdash \ell' : \alpha) & \text{si } s\ell = \mathsf{in}_{\mathbb{L}}\ell' \\ \emptyset & \text{sinon} \end{cases}$

Lemme 3.3.26

Acc$(\cdot : \theta)$ est une fonction d'accessibilité, et si $\theta <: \theta'$, alors

$$\mathrm{Acc}(\cdot : \theta) <: \mathrm{Acc}(\cdot : \theta')$$

3.4 Exemples

3.4.1 La suite de Meyer-Sieber

Les exemples suivants sont tirés de [MS88].

Meyer-Sieber 1, ou le Garbage Collector

Si x n'est pas libre dans M, et si $\Delta;\Gamma \vdash M : \mathbf{T}(A)$, alors

$$\Gamma \vdash \mathsf{let}\ x = \mathsf{ref}\ V\ \mathsf{in}\ M =_{\mathrm{ctx}} M : \mathbf{T}(A)$$

On prouve que $[\![\mathsf{let}\ x = \mathsf{ref}\ V\ \mathsf{in}\ M]\!]$ et $[\![M]\!]$ sont reliés par $\mathcal{R}_{\Gamma \vdash \mathbf{T}(A)}(\Delta\top)$ ($\top$ est la relation totale), et on conclut en utilisant le théorème 3.3.22. Soit $\Delta' r' \rhd \Delta\top$ un paramètre et $(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta' r')$. On veut prouver que $([\![\mathsf{let}\ x = \mathsf{ref}\ V\ \mathsf{in}\ M]\!]\rho_1, [\![M]\!]\rho_2) \in \mathcal{R}_{\mathbf{T}(A)}(\Delta' r')$. Soient $\Delta'' r'' \rhd \Delta' r'$, $(k_1, k_2) \in \mathcal{R}_{A^\top}(\Delta'' r'')$ et $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$.
Il faut prouver

$$[\![\mathsf{let}\ x = \mathsf{ref}\ V\ \mathsf{in}\ M]\!]\rho_1 k_1 S_1 = [\![M]\!]\rho_2 k_2 S_2$$

Pour $\ell \notin supp(k_1) \cup supp(S_1) \cup supp(V)$

$$[\![\mathsf{let}\ x = \mathsf{ref}\ V\ \mathsf{in}\ M]\!]\rho_1 k_1 S_1 = [\![M]\!]\rho_1 k_1 S_1[\ell \to [\![V]\!]\rho_1]$$

car x n'est pas libre dans M. Comme on peut choisir n'importe quel ℓ, on en choisit également hors de $A_{id''_\Delta \otimes r''}(S_i)$ pour $i = 1, 2$. Par le lemme fondamental, $[\![M]\!]$ est invariant par $\mathcal{R}_{\Gamma \vdash \mathbf{T}(A)}(\Delta\top)$, donc si on prouve que $(S_1[\ell \to [\![V]\!]\rho_1], S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$, on aura terminé.

D'abord, par définition des fonctions d'accessibilté

$$A_{id''_\Delta \otimes r''}(S_1[\ell \to [\![V]\!]\rho_1]) = A_{id''_\Delta \otimes r''}(S_1)$$

et donc $S_1[\ell \to [\![V]\!]\rho_1] \sim S_1 : A_{id''_\Delta \otimes r''}$. Comme $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$, par définition d'une relation finitaire entre états, $(S_1[\ell \to [\![V]\!]\rho_1], S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$, ce qui conclut la preuve.

Meyer Sieber 2

On rappelle le terme M :

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{ref}\ \underline{0}\ \mathsf{in}\\ f\ ();\\ \mathsf{let}\ z = !x\ \mathsf{in}\\ \mathsf{if}\ z\ \mathsf{then}\ \Omega\ \mathsf{else}\ \mathsf{val}\ ()\ \mathsf{fi}\end{array}$$

Prouvons que $([\![;\Gamma \vdash M : \mathbf{T}(\mathsf{U})]\!], \bot) \in \mathcal{R}_{\mathbf{T}(\mathsf{U})}(\emptyset\top)$, où $\Gamma = f : \mathsf{U} \to \mathbf{T}(\mathsf{U})$. On concluera alors grâce au théorème 3.3.22. Soient $\Delta' r' \rhd \Delta r$ $(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta r)$, $(k_1, k_2) \in \mathcal{R}_{\mathsf{U}^\top}(\Delta' r')$ et $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r')$. Il faut montrer que $[\![M]\!]\rho_1 k_1 S_1 = \bot$. On a $[\![M]\!]\rho_1 k_1 S_1 = \rho_1(p) * k_1' S_1'$ où, pour un ℓ bien choisi,

$$\begin{array}{rcl} k_1' & = & \lambda S *. \left\{ \begin{array}{ll} \bot & \text{si } S\ell = \mathsf{in}_{\mathbb{Z}} 0 \\ kS* & \text{sinon} \end{array} \right. \\ S_1' & = & S_1[\ell \to 0] \end{array}$$

Soit $r'' = r' \otimes \langle \{(S_1, S_2) \mid S_1\ell = \mathsf{in}_{\mathbb{Z}} 0\}, \{\ell\} \rangle$. On vérifie facilement que

$$\begin{array}{l}(k_1', \bot) \in \mathcal{R}_{\mathsf{U}^\top}(\Delta' r'')\\ (S_1', S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r'')\end{array}$$

Comme $(\rho_1(p), \rho_2(p)) \in \mathcal{R}(\Delta' r')$ et $\Delta' r'' \rhd \Delta' r'$,

$$[\![M]\!]\rho_1 k_1 S_1 = \rho_1(p) * k_1' S_1' = \rho_2(p) * \bot S_2$$

On conclut alors en rappelant que les programmes sont stricts, et donc que $\rho_2(p) * \bot S_2 = \bot$.

Meyer-Sieber 3

Égalité dans les domaines FM.

Meyer-Sieber 4,5,6

On présente l'exemple 6 (les exemples 4 et 6 sont similaires). Soit M le terme

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{ref}\ \underline{0}\ \mathsf{in}\\ \mathsf{let}\ almost_add2 = \lambda z.\mathsf{if}\ z = x\\ \qquad\qquad\qquad\qquad \mathsf{then}\ x := 1\\ \qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{let}\ y = !x\ \mathsf{in}\ \mathsf{let}\ y' = y + 2\ \mathsf{in}\ x := y' \mathsf{fi}\ \mathsf{in}\\ p(almost_add2);\\ \mathsf{let}\ y = !x\ \mathsf{in}\\ \mathsf{if}\ y\ \mathsf{mod}\ 2 = 0\ \mathsf{then}\ \Omega\ \mathsf{else}\ \mathsf{val}\ ()\ \mathsf{fi}\end{array}$$

On prouve que $(\llbracket;\Gamma \vdash M : \mathbf{T}(\mathsf{U})\rrbracket, \lambda\rho ks.\bot) \in \mathcal{R}_{\Gamma\vdash\mathbf{T}(\mathsf{U})}(\emptyset\mathbb{T})$. On a, pour un ℓ bien choisi :

$$\llbracket M\rrbracket\rho_1 k_1 S_1 = \rho_1(p) f \left(\lambda ks. \left\{ \begin{array}{ll} * & \text{si } S\ell = \mathsf{in}_{\mathbb{Z}} n \text{ pour un } n \text{ impair} \\ \bot & \text{sinon} \end{array} \right.\right) S_1[\ell \to 0]$$

où

$$fzkS = \left\{ \begin{array}{ll} kS[\ell \to 1]* & \text{si } z = \ell \\ kS[\ell \to n+2]* & \text{si } z \neq \ell \text{ et } s\ell = \mathsf{in}_{\mathbb{Z}} n \\ \bot & \text{sinon} \end{array} \right.$$

Soient $\Delta' r' \rhd \Delta r$ deux paramètres, $(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta r)$, $(k_1, k_2) \in \mathcal{R}_{\mathsf{U}^\top}(\Delta' r')$ et $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r')$. Soit $r'' = r' \otimes \langle \{(S_1, S_2) \mid S_1\ell, S_2\ell \text{ sont des entiers pairs}\} \cup \{\bot, \bot\}, \{\ell\}\rangle$. On va prouver que

$$(f, f) \in \mathcal{R}_{\mathsf{N}\ \mathrm{ref}\to\mathbf{T}(\mathsf{U})}(\Delta' r'')$$

Le reste suit naturellement.

Supposons que $\Delta^4 r^4 \rhd \Delta^3 r^3 \rhd \Delta' r''$, $(v_1^3, v_2^3) \in \mathcal{R}_{\mathsf{N}\ \mathrm{ref}}(\Delta^3 r^3)$, $(k_1^4, k_2^4) \in \mathcal{R}_{\mathsf{U}}(\Delta^4 r^4)$, $(s_1^4, s_2^4) \in \mathcal{R}_{\mathbb{S}}(\Delta^4 r^4)$. Comme $\Delta^4 r^4 \rhd \Delta' r''$, $(S_1, S_2) \in r^4 \subseteq r''$, donc pour $i = 1, 2$, il existe un état S_i tel que S_i contient un entier n_i. $v_1^3 = v_2^3$ est une adresse visible, donc, comme $A_{r^4} <: A_{r''}$, $\ell \in A_{r^4}(S_i)$, ce qui amène $v_i^3 \neq \ell$. On obtient

$$f v_i k_i S_i = k_i S_i[\ell \to n_i + 2]$$

Comme $(S_1[\ell \to n_i + 2], S_2[\ell \to n_i + 2]) \in \mathcal{R}_{\mathbb{S}}(\Delta^4 r^4)$ (notre invariant est préservé et, par non-interférence, les autres parties de l'invariant aussi) :

$$f v_1 k_1 S_1 = k_1 S_1[\ell \to n_1 + 2] = k_2 S_2[\ell \to n_2 + 2] = f v_2 k_2 S_1$$

Meyer-Sieber 7

On définit M_i :

$$\begin{array}{l} \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\ \mathsf{let}\ add_i = \lambda u. \\ \qquad \mathsf{let}\ y = !x\ \mathsf{in} \\ \qquad \mathsf{let}\ y' = y + i\ \mathsf{in} \\ \qquad x := y' \\ \mathsf{in} \\ p\ add_i \end{array}$$

On prouve que M_1 et M_2 sont équivalents pour toutes les relations Δr. Soient $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta r)$, $(k_1, k_2) \in \mathcal{R}_{\mathsf{U}^\top}(\Delta r)$. On a, pour un nouveau ℓ_x :

$$\llbracket M_i\rrbracket\rho_i k_i S_i = \rho_i(p) add_i k_i (S_i[\ell \to 0])$$

où

$$add_i k s * = \begin{cases} k(s[\ell \to n+i])* & \text{si } s\ell = \mathsf{in}_{\mathbb{Z}} n \\ \bot & \text{sinon} \end{cases}$$

On peut vérifier que si r' est l'invariant "ℓ contient un entier", et $r^2 = r \otimes r'$, alors $(add_1, add_2) \in \mathcal{R}_{\mathsf{U} \to \mathbf{T}(\mathsf{U})}(\Delta r^2)$, et $(S_1[[\ell \to 0], S_2[[\ell \to 0]) \in \mathcal{R}_{\mathbb{S}}(\Delta r^2)$. Comme $\Delta r^2 \rhd \Delta r$, $(k_1, k_2) \in \mathcal{R}_{\mathsf{U}^\top}(\Delta r^2)$, on obtient $[\![M_1]\!]\rho_1 k_1 S_1 = [\![M_2]\!]\rho_2 k_2 S_2$.

3.4.2 Typage généralisé

Comme illustration des types d'état généralisés, on prouve que le programme M défini par

$$\begin{array}{l} \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\ \mathsf{let}\ y = \mathsf{ref}\ x\ \mathsf{in} \\ p\ x; \\ \mathsf{let}\ z = !y\ \mathsf{in} \\ \mathsf{let}\ b = (z = x)\ \mathsf{in} \\ \mathsf{if}\ b\ \mathsf{then}\ \Omega\ \mathsf{else}\ \mathsf{val}\ ()\ \mathsf{fi} \end{array}$$

diverge, en utilisant un invariant qui pointe vers une adresse visible, en montrant que $([\![M]\!], \bot) \in \mathcal{R}_{\Gamma \vdash \mathbf{T}(\mathsf{U})}(\Delta r)$ pour tout Δr. Soit $\Delta^2 r^2 \rhd \Delta' r' \rhd \Delta r$, $(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta' r')$, $(k_1, k_2) \in \mathcal{R}_{\mathsf{U}^\top}(\Delta^2 r^2)$ et $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta^2 r^2)$. Il faut prouver que $[\![M]\!]\rho_1 k_1 S_1 = \bot$.

Pour tout nouveau ℓ_x, ℓ_y,

$$[\![M]\!]\rho_1 k_1 S_1 = \rho_1(p)\ell_x(\lambda s.\lambda_.\text{si } \ell_y = \mathsf{in}_{\mathbb{L}}\ell_x \text{ alors } \bot \text{ sinon } k_1 s *) s_1'$$

où $S_1' = S_1[\ell_y \to \ell_x, \ell_x \to 0]$. L'idée est que la valeur contenue dans ℓ_y quand on atteint la continuation est ℓ_x. Le problème est que ℓ_x est donnée à p, et devient donc visible. Grâce au nouveau type $\mathbb{T}$, on peut exprimer que l'invariant ne considère que l'adresse contenue dans ℓ_y, mais pas ce qui y est stocké, en donnant à ℓ_y le type $\mathbb{T}$ ref.

Formellement, on prouve que

$$(\lambda s.\lambda_.\text{si } \ell_y = \mathsf{in}_{\mathbb{L}}\ell_x \text{ alors } \bot \text{ sinon } k_1 s *, \lambda s.\lambda_.\bot) \in \mathcal{R}_{\mathsf{U}^\top}(\Delta^3 r^3)$$

où $\Delta^3 = \Delta^2[\ell_x \to \mathsf{N}\ \mathrm{ref}]$ et $r^3 = r^2 \otimes dr$, où

$$dr = \langle \{(S_1, S_2) \mid S_1 \ell_y = \mathsf{in}_{\mathbb{L}}\ell_x\}, \{\ell_y : \mathbb{T}\ \mathrm{ref}\} \rangle$$

$\Delta^3 r^3$ est plus grand que $\Delta^2 r^2$, donc $(\rho_1(p), \rho_2(p)) \in \mathcal{R}_{\mathsf{U}}(\Delta^3 r^3)$ (affaiblissement). $(S_1', S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta^3 r^3)$, car ils sont évidemment dans id_{Δ^3} et dans r^3, et car

$\mathrm{Acc}(s_1' \vdash \ell_y : \mathbb{T}\,\mathrm{ref}) = \{\ell_y\}$ n'atteint pas ℓ_x. On conclut en rappelant que $\rho_2(p)$ est stricte :

$$\begin{array}{rcl} [\![M]\!]\rho_1 k_1 S_1 & = & \rho_1(p)\ell_x(\lambda s.\lambda_.\text{si } \ell_y = \mathsf{in}_{\mathbb{L}}\ell_x \text{ alors } \bot \text{ sinon } k_1 s *)s_1' \\ & = & \rho_2(p)\ell_x(\lambda s.\lambda_.\bot)S_2 \\ & = & \bot \end{array}$$

3.4.3 Memoisation

Cet exemple est tiré de [PS98]. On considère le terme M :

$\lambda f.$let $a =$ ref 0 in
  let $z = f\ 0$ in
  let $r =$ ref z in
  val $(\lambda x.$let $v = !a$ in
    let $d = v - x$ in
    if d then val $()$ else $a := x$; let $r' = f\ x$ in $r := r'$ fi;
    $!r)$

On prouve que si φ se comporte comme une fonction "pure", i.e. s'il existe une fonction $\bar{\varphi} : \mathbb{Z} \to \mathbb{Z}$ telle que pour tout k, S, $\varphi k S = kS(\bar{\varphi}n)$, alors $([\![M]\!]\rho\varphi, \mathsf{val}\ \varphi) \in \mathcal{R}_{\mathbf{T}((\mathsf{N}\to\mathbf{T}(\mathsf{N})))}(\Delta r)$ pour tout Δr.

Soient $(k_1, k_2) \in \mathcal{R}_{\mathsf{N}\to\mathbf{T}(\mathsf{N})^\top}(\Delta r)$ et $(S_1, S_2) \in id_\Delta \otimes r$. D'abord, si ℓ_a, ℓ_r sont de nouvelles adresses :

$$[\![M]\!]\rho\varphi k S = kS[\ell_a \to 0, \ell_r \to \phi(0)]\varphi'$$

où $\varphi' = [\![\lambda x. \ldots .]\!]\rho[f \to \varphi, a \to \ell_a, r \to \ell_r]$. On définit

$$dr = \langle \{(S_1, S_2) \mid \exists n \in \mathbb{Z}, S_1\ell_a = \mathsf{in}_{\mathbb{Z}} n \ \wedge\ S_1\ell_r = \mathsf{in}_{\mathbb{Z}}\bar{\varphi}(n)\}, \mathrm{Acc}_{\{\ell_a:\mathsf{N},\ell_r:\mathsf{N}\}}\rangle$$

Il est alors facile de vérifier que $(\varphi', \varphi) \in \mathcal{R}_{\mathsf{N}\to\mathbf{T}(\mathsf{N})}(\Delta(r \otimes r_0))$ et $(S_1[\ell_a \to 0, \ell_r \to \phi(0)], S_2) \in id_w \otimes r \otimes r_0$. Comme $\Delta(r \otimes r_0) \rhd \Delta r$:

$$[\![M]\!]\varphi k_1 S_1 = k_1 S_1[\ell_a \to 0, \ell_r \to \phi(0)]\varphi' = k_2 S_2 \varphi = (\mathsf{val}\ \varphi) k_2 S_2$$

3.4.4 Réversibilité des changements (snapback)

On reprend une équivalence de [OR00] On voudrait prouver que le terme M

let $x =$ ref 0 in
  $p(\lambda u.x := 1;\ 0)$;
let $y = !x$ in
if y then val $()$ else Ω fi

est équivalent dans le modèle au terme N

$$p\ (\lambda u.\Omega)$$

En effet, intuitivement, soit p utilise son argument, et alors dans le premier cas x ne contient plus 0 et M diverge, comme N, soit p ne l'utilise pas, et alors, si p termine, x contient toujours 0, donc M, comme N termine si et seulement si p termine.

Malheureusement, dans notre modèle, la fonction suivante est acceptable :

$$\mathsf{snapback} = \lambda f.\lambda k.\lambda S.f\ *\ (\lambda S'.\lambda v.k\ S\ v)\ S$$

Elle exécute son argument f sur l'état S, et récupère (si f termine) un nouvel état S'. Elle exécute alors sa continuation k, mais avec l'état d'origine S.

Avec $p = \mathsf{snapback}$, le terme M termine alors que le terme N diverge : snapback utilise son argument, donc N ne termine pas, mais snapback annule tous les changements que son argument a pu faire, donc x est remis à 0, ce qui fait que M termine.

En fait snapback combine deux comportements interdits : elle duplique l'état S et elle n'utilise pas l'état S'. Logiquement, cela correspond à la contraction et à l'affaiblissement. S'il est compliqué de supprimer les éléments du modèle qui utilisent la contraction, il devrait en revanche être assez facile de supprimer les affaiblissements : cela correspond à demander que les continuations soient strictes en leur paramètre S. Ainsi, O'Hearn et Reynolds [OR00] et Pitts [Pit97], grâce à un simple soulèvement de l'ensemble des états, parviennent à éliminer les affaiblissements. Comme toutes les fonctions du type de snapback connues utilisent à la fois affaiblissement et contraction, cela suffit à améliorer très largement le modèle.

Dans notre modèle, nous avons essayé d'appliquer cette technique en soulevant directement $\mathbb{S}$, mais les résultats ont été moins bons que prévu : la fonction snapback était bien éliminée, les théorèmes restaient vrais, mais beaucoup d'autres équivalences plus faciles devenaient fausses. Il faudrait sans doute utiliser des états partiels $\mathbb{S} = \mathbb{L} \to (\mathbb{Z} + \mathbb{L})_\perp$, mais cela entraine des complications techniques dans les preuves sur les relations logiques, qui nous ont fait préférer les états totaux. Éliminer snapback de notre modèle reste donc un problème ouvert.

3.4.5 Protocoles cryptographiques

Nous pouvons également prouver des exemples inspirés des travaux de Sumii et Pierce [SP01] sur les relations logiques et l'encryption. L'idée, et nous ne prétendons pas que c'est un modèle convaincant de la cryptographie, est

que les messages sont envoyés à travers des adresses cachées. Le cryptage est une écriture, et le décryptage une lecture. Les adresses visibles représentent les clés connues du contexte, et les invariants les clés privées. Les clés publiques (appariées avec des clés privées) sont représentées par des fonctions qui écrivent dans l'adresse secrète.

Pour notre exemple, nous utiliserons le langage étendu avec des types produits, qui peuvent être écrits en mémoire. Tous les théorèmes énoncés plus haut restent valides pour ce langage, comme nous l'avons montré dans [BL05]. La paire est notée $\langle a, b\rangle$, et on autorise la reconnaissance de motif dans les fonctions : $\lambda\langle a, b\rangle.P$.

Dans le protocole (simpliste) suivant, A (la première fonction) envoie un message à B (la deuxième fonction) qui contient une nouvelle clé. B lit ce message, et envoie un entier i en utilisant la clé reçue. Le programme M_i s'écrit :

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\\ \mathsf{let}\ k_b = \mathsf{ref}\ x\ \mathsf{in}\\ \mathsf{let}\ \mathrm{cipher} = \mathsf{val}\ \lambda\langle k, n\rangle.k := n\ \mathsf{in}\\ \mathsf{let}\ \mathrm{decipher} = \mathsf{val}\ \lambda k.!k\ \mathsf{in}\\ \mathsf{val}\ \langle \lambda().\mathsf{let}\ k_a = \mathsf{ref}\ 0\ \mathsf{in}\ \mathrm{cipher}\ \langle k_b, k_a\rangle,\\ \qquad \lambda().\mathsf{let}\ k = \mathrm{decipher}\ k_b\ \mathsf{in}\ \mathrm{cipher}\ \langle k, i\rangle\rangle\end{array}$$

Pour prouver que l'information envoyée de B à A est bien secrète, on prouve que les M_i sont observationellement équivalents. Il suffit en fait de montrer que $M_1 \approx M_2$.

Un calcul facile donne, pour de nouvelles adresses ℓ_x et ℓ_b :

$$[\![M_i]\!]kS = kS[\ell_b \to \ell_x \to 0]\langle \phi^A, \phi_i^B\rangle$$

où

$$\begin{array}{l}\phi^A = \lambda * kS.kS[\ell_d \to \ell_a \to 0] * \quad \text{pour un nouveau } \ell_a\\ \phi_i^B = \lambda * kS. \left\{ \begin{array}{ll} kS[\ell \to i]* & \text{si } S\ell_b = \mathsf{in}_{\mathbb{L}}\ell \\ \bot & \text{sinon}\end{array}\right.\end{array}$$

Soit Δr un paramètre, $(k_1, k_2) \in \mathcal{R}_{(B\times B)^\top}(\Delta r)$ et $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta r)$, où $B = \mathsf{U} \to \mathbf{T}(\mathsf{U})$ est le type des processus A et B. L'invariant dr dit que ℓ_b contient une référence vers un entier, ce qui rend ℓ_b et, plus tard, ℓ_a secrètes. Soit $r' = r \otimes dr$, et $S_i' = S_i[\ell_d \to \ell_k \to 0]$. Bien sûr, $(S_1', S_2') \in \mathcal{R}_{\mathbb{S}}(\Delta r')$. Si on montre que (ϕ^A, ϕ^A) et (ϕ_i^B, ϕ_j^B) sont dans $\mathcal{R}_B(\Delta r')$, on aura terminé.

Soit $\Delta'' r'' \rhd \Delta r'$, $(k_1'', k_2'') \in \mathcal{R}_{\mathsf{U}^\top}(\Delta'' r'')$ et $(S_1'', S_2'') \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$. On écrit $r'' = r \otimes dr \otimes dr'$. On peut choisir une nouvelle adresse ℓ_k telle que, pour $i = 1, 2$:

$$\phi^A * k_i'' S_i'' = k_i'' S_i''[\ell_b \to \ell_a \to 0]*$$

Il est facile de vérifier que les nouveaux états sont encore reliés par $\mathcal{R}_{\mathbb{S}}(\Delta'' r'')$: ils sont dans dr, les autres parties de la relation $(id_{\Delta''}, r)$ sont préservées grâce à la condition de séparation, car les seules adresses modifiées sont ℓ_b (qui est dans $A_{dr} S_i$) et ℓ_a (qui est nouvelle), et parce qu'il n'y a pas de nouveau pointeur croisé. On obtient $(\phi^A, \phi^A) \in \mathcal{R}_B(\Delta r')$.

La même propriété est vérifiée par ϕ_i^B et ϕ_j^B car la relation dr exprime seulement que ℓ_k contient un entier, mais ne dit pas lequel. On obtient donc bien l'équivalence attendue entre les M_i et M_j.

Par contre, si la clé est publique, alors le secret est brisé : le contexte peut envoyer un message à B (se faisant, de ce fait, passer pour A) en lui donnant une adresse qu'il peut ensuite lire pour récupérer le message i. Formellement, on donne au contexte (comme un troisième élément) la fonction

$$\text{public_cipher} = \lambda n.\text{cipher}\ \langle k_b, n \rangle$$

public_cipher ne respecte pas la relation $\mathcal{R}_{\mathbb{S}}(\Delta r')$:

$$\text{public_cipher}\ \ell k S = k S[\ell_b \to \ell] *$$

ℓ_b pointe bien vers une référence sur un entier après exécution, donc dr est préservé, mais cette référence est ℓ qui, elle, est visible, donc la condition de séparation entre id_Δ et dr est violée.

3.5 Conclusion

On a vu dans ce chapitre un modèle pour un langage avec allocation dynamique, et une méthode pour prouver des équivalences grâce à ce modèle. L'utilisation des domaines FM, avec une action de permutation des noms (des adresses) est nouvelle dans ce contexte. Elle permet d'éliminer les observations grossièrement non définissables, tout en conservant une présentation accessible. Les relations paramétriques, plus classiques, permettent en revanche de mettre en valeur des comportements spécifiques.

Finalement, notre modèle, accompagné des relations logiques, se situe dans une moyenne honorable : il est loin d'être complet (en particulier, la présence de snapback est problématique), mais, comparé aux modèles dénotationnels précédents [Lev02, Sta94, RY04] comme aux modèles de termes [PS98, Pit02], le modèle de base est élémentaire. Les relations logiques sont bien sûr plus difficiles à manier, mais la paramétricité est inévitablement accompagnée d'une certaine lourdeur. Sur ce terrain, notre construction est comparable aux travaux antérieurs similaires [RY04, OR00, OT95].

Les extensions possibles sont de deux ordres. D'abord, il devrait être possible d'éliminer snapback en utilisant un soulèvement du domaine des états, comme chez Pitts [Pit02] ou O'Hearn et Tennent [OR00]. D'autre part, le langage peut être enrichi : outre les types produit et les types somme, que nous avons déjà examinés dans [BL05] (nous les avons omis ici car ils ne présentent pas de difficultés particulières, et pour garder un langage proche de PCF), les types inductifs ne devraient pas poser de gros problèmes : la notion d'accessibilité se prête assez bien à ce genre de structures.

Un des buts de ces travaux était d'étendre les modes de preuve de [Ben04] aux langages de bas niveau utilisés lors de la compilation de langages fonctionnels : autoriser les pointeurs sur des fonctions est alors indispensable, car on construit des clôtures "à la main". Malheureusement, la présence de types récursifs et l'enregistrement de fonctions semblent rendre la construction beaucoup plus ardue : en appliquant directement les techniques décrites plus haut, les paramètres ne forment plus un bon ordre. Il faudrait procéder avec plus de prudence, et les travaux de Pitts sur les points fixes dans des catégories devraient se révéler utiles.

Dans le cadre plus précis de cette thèse, ces travaux jettent une lumière nouvelle sur la notion d'observation : au contraire de la partie précédente, nous n'avons pas cherché à éliminer les problèmes de séquentialité (d'ailleurs, dans sa forme actuelle, notre modèle contient des contextes qui lancent des évaluations parallèles), mais nous nous sommes plutôt concentrés sur les aspects reliés à la présence d'un état dont la configuration est dynamique.

Les observations interdites sont d'un autre ordre : alors que dans les cas précédents, il s'agissait de choses intrinsèques au modèle (comme les opérateurs de parallélisme), qui n'avaient pas de contrepartie syntaxique, on veut ici interdire aux contextes d'examiner une partie de l'état qui existe bel et bien, mais à laquelle ils n'ont pas le droit d'accéder. Ce glissement est accompagné d'un changement de vocabulaire : on a beaucoup parlé de partie visible ou publique, et d'invariants ou de partie privée ou secrète. Cela rappelle bien sûr l'analyse de flot : bien que nos résultats ne soient pas directement applicables, nous avons présenté un encodage (ni réaliste ni pratique) qui nous a permis de prouver un petit exemple de "protocole" cryptographique. Les travaux de Sumii et Pierce [SP01] laissent espérer des résultats beaucoup plus intéressants.

Chapitre 4

Coût en sémantique dénotationnelle

4.1 Utilisation de ressources

Dans cette partie, on s'intéresse à l'étude du coût en sémantique dénotationnelle. Contrairement aux parties précédentes, qui visaient à améliorer les modèles pour se rapprocher de la complète adéquation, c'est-à-dire réduire les possibilités de la sémantique, pour rapprocher son pouvoir expressif des observations syntaxiques, nous allons ici étendre des modèles pour représenter une propriété non observable par les contextes : la longueur du calcul. En effet, la seule propriété observable de cet ordre est la terminaison : un contexte pourra distinguer un terme qui termine d'un terme qui ne termine pas, mais pas deux termes qui renvoient la même valeur en prenant plus ou moins de temps.

Cette étude ne prétend pas définir une notion réaliste de temps de calcul, mais tend plus modestement à montrer qu'on peut enrichir un modèle pré-existant avec des informations sur la longueur du calcul, sans sacrifier sa précision : ainsi, le modèle de jeux enrichi est, comme le modèle d'origine, complètement adéquat[1]. Nous utiliserons pour cela une monade, qui permet, comme souvent (cf. par exemple [Mog91, PJW93]) de séparer efficacement les problématiques : toute la partie "langage" sera interprétée par le modèle de départ, et les calculs relatifs au temps seront effectués par la monade.

Nous ne nous intéresserons pas à la complexité, mais plutôt au coût. La complexité temporelle s'intéresse à la relation entre la taille des entrées et la longueur du calcul. Comme souvent en sémantique dénotationnelle, les entiers sont un domaine plat, et ils n'ont pas de taille associée : il faudrait instrumenter lourdement un modèle dénotationnel classique pour y intégrer

[1] Il y a eu une perte, tout de même, en ce que la propriété de définissabilité (des éléments compacts) n'est plus valide.

une notion raisonnable de complexité, et il semble difficile de présenter une méthode générique pour y parvenir.

Au contraire, le coût d'un programme est la quantité exacte d'une certaine ressource consommée lors du calcul : il faut donc définir un modèle de coût, c'est-à-dire la ressource mesurée, et ce qui la consomme. Par exemple, pour comparer des algorithmes de tri, on utilise en général le nombre de comparaisons. Dans notre étude, nous compterons le nombre de tics d'horloge nécessaire, sous l'hypothèse simplificatrice que chaque étape de réduction consomme un tic.

Nous raisonnerons en plusieurs étapes. D'abord, il faut définir une notion de temps de calcul, et pour cela nous allons définir une machine, inspirée de la machine de Krivine [Kri04], et prouver que la sémantique à grand pas de PCF (définie dans les préliminaires) et la sémantique à petits pas (définie par la machine) coincident. À partir de là, deux possibilités s'offrent à nous :

- on peut traduire les termes du langage dans le langage étendu avec une opération tick, et étendre le modèle en ajoutant une interprétation pour tick,
- on peut aussi interpréter directement le langage d'origine, en modifiant plus en profondeur le modèle.

Chacune des méthodes a ses avantages. La deuxième possiblité pourrait être beaucoup plus précise, mais il semble difficile de trouver les contraintes sur les domaines qui vont correspondre à la notion de coût choisie, qui est complètement arbitraire. Par contre, on peut espérer mettre en relief une notion de coût beaucoup moins arbitraire, puisqu'elle dérive d'une construction mathématique. On présentera le modèle des stratégies semi-alternantes, qui ne modélise pas un coût syntaxique mais la longueur des parties de la sémantique des jeux.

La première solution a pour elle la simplicité et la clarté : en faisant apparaître les tics dans la syntaxe, on a bien distingué ce qui relève du temps de calcul de ce qui relève du comportement extensionnel, et les modifications à apporter au modèle sont mineures. On perd par contre beaucoup de précision lors de la première traduction : beaucoup de termes du langage étendu ne sont pas définissable par des termes du langage d'origine, ce qui, a priori, pénalise le modèle complet (l'ensemble des deux traductions). On verra que dans le cas des jeux, on perd bien sûr la définissabilité, mais qu'on garde malgré tout la complète adéquation (car justement le temps n'est pas observable dans le langage d'origine).

Nous donnerons une liste de conditions pour obtenir un modèle temporisé à partir d'un modèle de PCF en utilisant la première méthode : le point central est l'existence d'une monade telle que la catégorie de Kleisli soit cartésienne fermée, et d'un morphisme tick qui commute avec les fonctions

entre les entiers. Nous présenterons ensuite une monade particulière, à savoir la monade $T \Rightarrow -$, ou plutôt la comonade $T \times -$. Comme application, nous construirons un modèle d'espace de cohérence, et un modèle de jeux complètement adéquat. Finalement, nous présenterons une nouvelle classe de stratégies en sémantique des jeux, les stratégies semi-alternantes, qui permettent de mesurer la longueur d'une partie.

La communauté de la sémantique des langages de programmation s'est concentrée sur le problème de la complète adéquation, et moins de travaux ont été réalisés sur des propriétés non observables. L'analyse quantitative, ou la mesure du temps de calcul, a été lancée par, entre autres, Bjerner, Holström et Wadler. Sands [San90a, San90b, San95, San98], et Moran [MS00], ont généralisé leurs résultats aux fonctions d'ordre supérieur, et ils ont introduit la notion d'amélioration (improvement). Guy Blelloch et John Greiner [BG95, Gre97] ont aussi caractérisé la complexité par des méthodes opérationnelles, en ajoutant des modèles de coût à la sémantique. Alan Jeffrey [Jef92] a exploré le cas des langages concurrents.

La différence majeure avec ces travaux réside dans l'utilisation de la sémantique dénotationnelle. De ce point de vue, notre travail est plus proche de ceux d'Escardo [Esc98] ou de Ghica [Ghi05]. Il s'en distingue néanmoins par l'utilisation d'une monade : pour propager les informations de temps par composition, Escardo utilise des domaines spéciaux (les espaces ultramétriques), et Ghica instrumente la sémantique des jeux (en ajoutant des jetons). Notre construction, au contraire, peut être appliquée facilement à n'importe quelle catégorie où il est possible de "compter". L'utilisation d'une monade rappelle les travaux de Gurr [Gur91], même si son but était plutôt de comparer les complexités de différents langages. Plus récemment, Van Stone [VS03] a mené indépendamment une étude du coût en sémantique dénotationnelle en utilisant des monades. Ses résultats semblent plus aboutis, mais notre présentation est, à notre avis, à la fois plus générale et plus élégante.

Ces travaux devraient aussi être rapprochés des études sur les langages à complexité bornée : par exemple, Baillot et Mogbil [BM04] et Hoffman [Hof99] ont défini des variantes du λ-calcul où tous les programmes terminent en temps polynomial.

4.2 PCF temporisé

4.2.1 Machine

La sémantique de PCF (à grands pas) ne contient aucune information sur le temps d'exécution du programme. De plus, l'application est simulée

par une substitution qui peut-être très coûteuse. Pour examiner le temps de calcul d'un programme, on définit la sémantique opérationnelle en utilisant uniquement des opérations simples. Pour éviter la substitution, on utilise, comme dans toutes les implémentations des langages fonctionnels usuels, un environnement. Pour être vraiment réaliste, il aurait fallu de plus utiliser des indices de de Bruijn [dB72], mais on a gardé les noms de variables pour améliorer la lisibilité des définitions et des preuves.

États et réduction

Les états S de la machine sont composés de deux parties : une clôture "courante" C, qui contient le code à exécuter P et l'environnement E, et une continuation K, présentée comme une pile de contextes. L'environnement lie des clôtures aux variables, pour simuler l'appel par nom.

$$\begin{array}{rcl} S & ::= & \langle C, K \rangle \\ C & ::= & (P, E) \\ E & ::= & \{x_1 \to C_1, \ldots, x_n \to C_n\} \\ K & ::= & ([-]C \mid \mathsf{succ}\ [-] \mid \mathsf{pred}\ [-] \mid (\mathsf{if}\ [-]\ \mathsf{then}\ N\ \mathsf{else}\ P\ \mathsf{fi}, E))^* \end{array}$$

Les règles de réduction (figure 4.1) sont similaires, pour la partie λ-calcul, à la machine de Krivine [Kri04] : l'application pousse l'argument sur la pile en dupliquant l'environnement (pour former une clôture), et l'abstraction ajoute cette clôture à l'environnement courant (ce qui remplace la substitution). Quand on rencontre une variable, on va chercher sa valeur dans l'environnement. Les autres règles (pour les opérations sur les entiers) sont très simples : on empile un contexte qui indique le calcul qui reste à effectuer.

On pourra reprocher à ces règles de ne pas être complètement réalistes : si on implémentait les environnements comme des listes, alors le déréférencement ne serait pas unitaire, et si on les implémentait par des tableaux, alors il faudrait copier un environnement pour l'allonger (ce que fait, par exemple, le compilateur O'Caml).

On pourrait mettre des poids sur chaque réduction pour modéliser ces comportement, mais un compilateur efficace ne garde dans l'environnement que les variables utiles, ce qui compliquerait énormément la description. Bien qu'il ne semble pas y avoir d'obstacle théorique à cette extension, nous avons préféré garder la présentation la plus simple possible.

Il est aisé de vérifier que la réduction est déterministe, c'est-à-dire qu'à chaque étape, il y a au plus une règle qu'on peut appliquer. On appelle état terminal un état où aucune règle ne peut être appliquée.

$$\begin{array}{rcl}
\langle (x, E), K\rangle & \rightsquigarrow & \langle E(x), K\rangle \\
\langle (MN, E), K\rangle & \rightsquigarrow & \langle (M, E), [-](N, E)\cdot K\rangle \\
\langle (\mathsf{rec}\ f\ x = M, E), [-]C\cdot K\rangle & \rightsquigarrow & \langle (M, E[x \to C, f \to (\mathsf{rec}\ f\ x = M, E)]), K\rangle
\end{array}$$

$$\begin{array}{rcl}
\langle (\mathsf{succ}\ M, E), K\rangle & \rightsquigarrow & \langle (M, E), \mathsf{succ}\ [-]\cdot K\rangle \\
\langle (n, E), \mathsf{succ}\ [-]\cdot K\rangle & \rightsquigarrow & \langle (n+1, E), K\rangle
\end{array}$$

$$\begin{array}{rcl}
\langle (\mathsf{pred}\ M, E), K\rangle & \rightsquigarrow & \langle (M, E), \mathsf{pred}\ [-]\cdot K\rangle \\
\langle (n+1, E), \mathsf{pred}\ [-]\cdot K\rangle & \rightsquigarrow & \langle (n, E), K\rangle
\end{array}$$

$$\langle (\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P\ \mathsf{fi}, E), K\rangle \rightsquigarrow \langle (M, E), (\mathsf{if}\ [-]\ \mathsf{then}\ N\ \mathsf{else}\ P\ \mathsf{fi}, E)\cdot K\rangle$$
$$\langle (0, E), (\mathsf{if}\ [-]\ \mathsf{then}\ N\ \mathsf{else}\ P\ \mathsf{fi}, E')\cdot K\rangle \rightsquigarrow \langle (N, E'), K\rangle$$
$$\langle (n+1, E), (\mathsf{if}\ [-]\ \mathsf{then}\ N\ \mathsf{else}\ P\ \mathsf{fi}, E')\cdot K\rangle \rightsquigarrow \langle (P, E'), K\rangle$$

Fig. 4.1 – Règles de réduction de la machine en appel par nom.

Typage

Pour les preuves de correction et d'adéquation, on considèrera seulement les états bien formés, qu'on appelle états bien typés : il faut que la clôture et la continuation aient des types "opposés". Les règles d'inférences sont données en figure 4.2.

Définition 4.2.1 (États bien typés)

Un état $\langle C, K\rangle$ est bien typé s'il existe A tel que $C : A$ et $K : A^{\perp}$.

En examinant chaque règle, il est facile de se convaincre que la réduction préserve le bon typage :

Proposition 4.2.2

Les états bien typés sont clos par réduction.

Les états bien typés terminaux sont $\langle n, E, \epsilon\rangle$. Si S est un état bien typé, on écrit $S \rightsquigarrow^n m$ si $S \rightsquigarrow^n S'$, S' est terminal et $S' = \langle m, E, \epsilon\rangle$ pour un certain E. On écrit $S \rightsquigarrow^{\infty}$ si la réduction qui part de S est infinie. Comme $\rightsquigarrow$ préserve le bon typage, pour tout état bien typé S, soit il existe m, n tels que $S \rightsquigarrow^n m$, soit $S \rightsquigarrow^{\infty}$. On modélise le temps de calcul d'un programme M par le nombre de pas nécessaires pour réduire $\mathcal{S}(M)$ à une valeur m : c'est l'indice n de $\mathcal{S}(M) \rightsquigarrow^n m$.

$$\frac{\Gamma \vdash M : A \qquad E : \Gamma}{(M, E) : A}$$

$$\frac{C_1 : A_1 \ \ldots \ C_n : A_n}{\{x_1 \to C_1, \ldots, x_n \to C_n\} : (x_1 : A_1, \ldots, x_n : A_n)}$$

$$\epsilon : \mathsf{N}^\perp \qquad \frac{C : A \qquad K : B^\perp}{([-]C) \cdot K : (A \to B)^\perp}$$

$$\frac{K : \mathsf{N}^\perp}{(\mathsf{succ}\ [-]) \cdot K : \mathsf{N}^\perp} \qquad \frac{K : \mathsf{N}^\perp}{(\mathsf{pred}\ [-]) \cdot K : \mathsf{N}^\perp}$$

$$\frac{K : \mathsf{N}^\perp \quad E : \Gamma \quad \Gamma \vdash P : \mathsf{N} \quad \Gamma \vdash Q : \mathsf{N}}{(\mathsf{if}\ [-]\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}, E) \cdot K : \mathsf{N}^\perp}$$

Fig. 4.2 – Typage de la machine

Simulation

Il s'agit maintenant de prouver que la machine définie ci-dessus affine bien la sémantique à grand pas de PCF. Comme ces résultats ne sont pas critiques (tout ce qui suit se réfèrera seulement à la sémantique à petits pas), on ne détaillera pas toutes les preuves.

La sémantique à grand pas ne réduisant que les termes, il faut trouver quel terme est contenu dans un état : on substitue les termes contenus dans l'environnement dans le terme courant, et la continuation correspond à un contexte.

Définition 4.2.3 (Projection)

On définit la projection des clôtures sur les termes fermés C^* :

$$(M, \{x_1 \to C_1 \ldots x_n \to C_n\})^* = M[x_1 \to C_1^* \ldots x_n \to C_n^*]$$

Et la projection des continuations sur les contextes :

$$\begin{array}{rcl}
\epsilon^* & = & [-] \\
(([-]C) \cdot K)^* & = & K^*[[-]C^*] \\
(\mathsf{succ}\ [-] \cdot K)^* & = & K^*[\mathsf{succ}\ [-]] \\
(\mathsf{pred}\ [-] \cdot K)^* & = & K^*[\mathsf{pred}\ [-]] \\
((\mathsf{if}\ [-]\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}, E) \cdot K)^* & = & K^*[\mathsf{if}\ [-]\ \mathsf{then}\ (P, E)^*\ \mathsf{else}\ (Q, E)^*\ \mathsf{fi}]
\end{array}$$

Les états sont projetés sur des termes par $\langle C, K \rangle^* = K^*[C^*]$

Lemme 4.2.4

Si $\langle C, K\rangle$ est bien typé, alors $\vdash \langle C, K\rangle^* : \mathsf{N}$.

Démonstration : Par induction sur la preuve de bon typage de $\langle C, K\rangle$. ■

Lemme 4.2.5

Si $M \Downarrow M'$, et si $C^* = M$, alors il existe C' et n tels que $C'^* = M'$ et, pour toute continuation K,

$$\langle C, K\rangle \rightsquigarrow^n \langle C', K\rangle$$

Démonstration : Par induction sur la preuve de $M \Downarrow M'$. Le cas de base $n \Downarrow n$ est évident. Les autres cas sont similaires : on traitera pour l'exemple le cas $(M)N \Downarrow M'$.

La preuve de $(M)N \Downarrow M'$ nous donne M'' tel que $M \Downarrow \mathsf{rec}\ f\ x = M''$ (pour f et x bien choisis) et $M''[f \to \mathsf{rec}\ f\ x = M'', x \to N] \Downarrow M'$. En considérant les éventuelles premières étapes d'ouvertures de variables, on trouve n_0, M_0, N_0 et E_0 tels que $(M_0, E_0)^* = M$ et $(N_0, E_0)^* = N$ et

$$\langle C, K\rangle \rightsquigarrow^{n_0} \langle((M_0)N_0, E_0), K\rangle$$

L'étape suivante nous donne

$$\langle((M_0)N_0, E_0), K\rangle \qquad \rightsquigarrow \langle(M_0, E_0), [-](N_0, E_0)\cdot K\rangle$$

Par hypothèse d'induction, il existe C_1 et n_1 tels que $C_1^* = \mathsf{rec}\, f\, x = M''$ et

$$\langle(M_0, E_0), [-](N_0, E_0)\cdot K\rangle \rightsquigarrow^{n_1} \langle C_1, , [-](N_0, E_0)\cdot K\rangle$$

Encore une fois en considérant les ouvertures de variables, on trouve n_2, M_2 et E_2 tels que

$$\langle C_1, [-](N_0, E_0)\cdot K\rangle \rightsquigarrow^{n_2} \langle(\mathsf{rec}\ f\ x = M_2E_2), [-](N_0, E_0)\cdot K\rangle$$

L'étape suivante nous donne

$$\begin{aligned}&\langle(\mathsf{rec}\ f\ x = M_2, E_2), [-](N_0, E_0)\cdot K\rangle\\ &\quad\rightsquigarrow \langle(M_2, E_2[f \to (\mathsf{rec}\ f\ x = M_2, E_2), x \to (N_0, E_0)]), K\rangle\end{aligned}$$

Comme $(\mathsf{rec} f\, x = M_2, E_2)^* = M''$ et $(N_0, E_0)^* = N$, $(M_2, E_2[f \to (\mathsf{rec} f\, x = M_2, E_2), x \to (N_0, E_0)])^* = M''[f \to \mathsf{rec}\ f\ x = M'', x \to N]$. Par hypothèse d'induction, il existe n_3 et C_3 tels que $C_3^* = M'$ et

$$\langle(M_2, E_2[f \to (\mathsf{rec}\ f\ x = M_2, E_2), x \to (N_0, E_0)]), K\rangle \rightsquigarrow^{n_3} \langle C_3, K\rangle$$

Au total

$$\langle C, K\rangle \rightsquigarrow^{n_0+1+n_1+n_2+1+n_3} \langle C_3, K\rangle$$

et n_1, n_2 et n_3 sont bien indépendants de K. ■

Lemme 4.2.6

Si $S \rightsquigarrow S'$ et $S'^* \Downarrow m$, alors $S^* \Downarrow m$.

Démonstration : On examine chaque règle séparément. Pour toutes les règles d'introduction des contextes, on a bien sûr $S^* = S'^*$, donc la conclusion est trivialement vérifiée.

Pour les règles d'élimination, il faut remplacer le nouveau terme courant par l'ancien dans la preuve, et ajouter la règle correspondante. On obtient alors un nouvel arbre, qui est une preuve de $S^* \Downarrow m$. Les définitions formelles et les détails de la preuve sont très longs. On ne donnera qu'un exemple :

$$\langle (0, E), (\mathsf{if}\ [-]\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}, E') \cdot K \rangle \rightsquigarrow \langle (P, E'), K \rangle$$

L'arbre de preuve pour $\langle (P, E'), K \rangle^* \Downarrow m$ a la forme

$$\dfrac{\dfrac{\dfrac{\pi}{(P, E')^* \Downarrow p}}{\pi'[(P, E')^*]}}{\langle (P, E'), K \rangle^* \Downarrow m}$$

et le nouvel arbre de preuve est

$$\dfrac{\dfrac{\dfrac{0 \Downarrow 0 \qquad \dfrac{\pi}{(P, E')^* \Downarrow p}}{\mathsf{if}\ 0\ \mathsf{then}\ (P, E')^*\ \mathsf{else}\ (Q, E')^*\ \mathsf{fi} \Downarrow p}}{\pi'[\mathsf{if}\ 0\ \mathsf{then}\ (P, E')^*\ \mathsf{else}\ (Q, E')^*\ \mathsf{fi}]}}{\langle (0, E), (\mathsf{if}\ [-]\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}, E') \cdot K \rangle^* \Downarrow m}$$

■

Proposition 4.2.7 (Simulation)

Soit $\vdash M : \mathsf{N}$ un programme. $M \Downarrow m$ si et seulement s'il existe n tel que $M \rightsquigarrow^n m$, et $M \not\Downarrow$ si et seulement si $M \rightsquigarrow^\infty$.

Démonstration : Les deux propositions sont en fait équivalentes. On suppose que la première est vraie. Si $M \not\Downarrow$, alors pour aucun m, n, $M \rightsquigarrow^n m$: comme la réduction préserve le bon typage, et que les seuls états bien typés terminaux sont les $\langle (m, E), \epsilon \rangle$, on a $M \rightsquigarrow^\infty$. Réciproquement, si $M \rightsquigarrow^\infty$, alors pour aucun m, n, $M \rightsquigarrow^n m$, donc pour aucun m, $M \Downarrow m$: par définition, $M \not\Downarrow$.

On va donc seulement prouver la première équivalence. L'implication de gauche à droite est une conséquence immédiate du lemme 4.2.5. Pour la réciproque, on raisonne par récurrence sur n : si $n = 0$, on a $M = m$, et $m \Downarrow m$ est un axiome. Si $n > 0$, on applique le lemme 4.2.6 et l'hypothèse de récurrence. ■

Amélioration

La notion-clé de l'étude des coûts est l'amélioration, introduite par Sands [San98] : un terme améliore un autre s'il produit le même résultat en moins d'étapes, quel que soit le contexte.

Définition 4.2.8 (Amélioration)

Soient P, Q deux termes tels que $\Gamma \vdash P, Q : A$. On dit que P améliore Q, et on note $P \lesssim Q$, si pour tout contexte $C : (\Gamma \vdash A)^{\perp}$,

$$\langle (C[Q], \emptyset), \epsilon \rangle \rightsquigarrow^{n} m \quad \Longrightarrow \quad \exists n' \leq n, \langle (C[P], \emptyset), \epsilon \rangle \rightsquigarrow^{n'} m$$

Développer une théorie syntaxique pour cette notion devient assez rapidement technique [San98]. On s'attachera donc à explorer quelques possibilités dénotationnelles. En particulier, on construira un modèle complètement adéquat pour l'amélioration en appel par nom en utilisant la sémantique des jeux.

4.2.2 PCF+tick

Sémantique à grands pas

On veut modéliser la réduction à petits pas $\rightsquigarrow$. La sémantique dénotationnelle est plus adaptée aux sémantiques à grands pas. On va donc "décorer" la sémantique de PCF pour indiquer le nombre d'étapes de réduction. Le nouveau prédicat sera $\Downarrow^{n}$. Pour bien distinguer le temps des opérations "réelles", on ajoute à PCF un opérateur dédié tick.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{tick}; M : A}$$

Les règles de réduction de PCF sont presque inchangées (on ajoute seulement le même indice n partout), par exemple

$$\frac{M \Downarrow^{n} m}{\mathsf{succ}\ M \Downarrow^{n} m + 1}$$

et on ajoute une règle pour tick :

$$\frac{M \Downarrow^{n} P}{\mathsf{tick}; M \Downarrow^{n+1} P}$$

On appelle ce langage PCF+tick.

Pour "coller" à la sémantique définie par $\rightsquigarrow$, on ajoute aux termes de PCF des tics bien placés. On note $[M]$ cette traduction :

- $[x] = \mathsf{tick}; x$,
- $[(P)Q] = \mathsf{tick}; \mathsf{tick}; ([P])[Q]$,
- $[\mathsf{rec}\ f\ x = M] = \mathsf{rec}\ f\ x = [M]$,
- $[\mathsf{succ}\ M] = \mathsf{tick}; \mathsf{tick}; \mathsf{succ}\ [M]$,
- $[\mathsf{pred}\ M] = \mathsf{tick}; \mathsf{tick}; \mathsf{pred}\ [M]$,
- $[\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}] = \mathsf{tick}; \mathsf{tick}; \mathsf{if}\ [M]\ \mathsf{then}\ [P]\ \mathsf{else}\ [Q]\ \mathsf{fi}$.

On pourrait prouver qu'il existe un E tel que $\langle (M, \emptyset), \epsilon\rangle \rightsquigarrow^n \langle (m, E), \epsilon\rangle$ si et seulement si $M \Downarrow^n m$. On prouvera en fait directement que la sémantique dénotationnelle définie à travers cette interprétation est correcte et adéquate.

Un modèle naïf

Pour définir la sémantique dénotationnelle de $\rightsquigarrow$, il suffit d'avoir un modèle de PCF "traditionnel" et de définir une interprétation de tick.

L'idée la plus simple est de coupler la sémantique avec un entier n. tick serait alors la fonction qui à $\langle n, x\rangle$ associe $\langle n+1, x\rangle$, alors que les autres opérations agiraient seulement sur x. Le problème est que les fonctions de type $A \to B$ seraient interprétées dans

$$N \times [\![A]\!] \longrightarrow N \times [\![B]\!]$$

La fonction qui à $\langle n, x\rangle$ associe $\langle 0, x\rangle$ est alors tout à fait envisageable. Elle est pourtant absurde : bien qu'elle calcule son argument, elle prétend ne pas avoir besoin de temps pour cela.

En fait, comme le temps n'est pas observable dans PCF, la fonction ne devrait pas avoir d'information sur le temps de calcul de son argument, mais seulement sur sa valeur : c'est typiquement une monade. L'avantage est que les fonctions ne pourront pas tricher, puisque la propagation des tics sera "câblée" dans la catégorie.

Par contre, en appel par nom, utiliser deux fois un argument requiert deux fois son calcul. Par exemple, si la fonction $\lambda x.\mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ x\ \mathsf{fi}$ associe m à $\langle n, m\rangle$ (elle renvoie à la même valeur en n étapes), et que $f : 1 \to N \times \mathsf{N}$ soit $\langle n', m\rangle$ (produit m en n' étapes), la composition devrait donner $\langle n + 2n', m\rangle$ au lieu de $\langle n + n', m\rangle$. Il apparaît ici que les catégories de fonctions ne pourront pas fournir de modèle correct : il va falloir utiliser des catégories plus complexes où il est possible de "compter" les utilisations de chaque argument. Le problème serait similaire si on voulait modéliser l'appel par necessité : on ne recalculerait pas plusieurs fois le même argument, mais, comme d'ailleurs pour le cas de l'appel par nom, on ne le calculerait pas du tout s'il n'est pas utilisé. Il faudrait donc une catégorie où on peut savoir si un argument est utilisé ou pas. Nous présenterons quelques pistes pour le cas

de l'appel par nécessité dans la conclusion de cette partie, mais le corps de ce travail s'attache à la modélisation de l'appel par nom.

Le plan est donc le suivant :
- choisir un modèle traditionnel de PCF,
- définir une monade pour représenter le temps,
- définir une opération tick,
- définir la sémantique de PCF (à travers la traduction $[-]$ dans PCF+tick),
- vérifier que la catégorie de Kleisli est bien un modèle de PCF.

On va en fait énoncer une liste d'axiomes sur la catégorie et la monade qui permettront de définir la sémantique et de prouver la correction et l'adéquation du modèle obtenu. Il suffira alors de vérifier ces axiomes.

4.2.3 Modèle

Description catégorique

On se donne une sous-catégorie $\mathcal{C}$ de CPO avec une monade forte $\langle T, \eta, \nu, t\rangle$.
On notera en gras les opérations dans la catégorie de Kleisli :
- la composition sera notée $\bullet$,
- la paire sera notée $\langle -, -\rangle$,
- les projections seront notées $\boldsymbol{\pi}_i$.

On suppose que les conditions suivantes sont vérifiées :
- $\mathsf{N}_\perp$ est un objet de $\mathcal{C}$ et $\mathcal{C}[1, T\mathsf{N}_\perp]$ contient une famille m_n pour m, n entiers naturels, avec $m_0 = \eta \circ m$,
- il existe un morphisme $\mathsf{tick}_\mathsf{N} : \mathsf{N}_\perp \to T\mathsf{N}_\perp$ tel que $\mathsf{tick}_\mathsf{N} \bullet x = m_n$ si et seulement si $n > 0$ et $x = m_{n-1}$ et pour tout morphisme $f \in C[\mathsf{N}_\perp, T\mathsf{N}_\perp]$ $f \bullet \mathsf{tick}_\mathsf{N} = \mathsf{tick}_\mathsf{N} \bullet f$,
- le produit $- \times -$ rend la catégorie de Kleisli cartésienne, avec $\langle f, g\rangle = \bar{t}^* \circ t \circ \langle f, g\rangle$ et $\boldsymbol{\pi}_i = \eta \circ \pi_i$,
- la catégorie de Kleisli est fermée, et on notera $\boldsymbol{\Lambda}$ et $\mathbf{V}$ les opérations de curryfication et de décurryfication, et on notera $\mathbf{ev}_{X,Y} = \mathbf{V}(\eta_{X \Rightarrow Y})$ (attention, $X \Rightarrow Y$ est l'objet de la catégorie $\mathcal{C}$, ce n'est pas $X \Rightarrow TY$).

On interprète un état de la machine $\langle C, K\rangle$ par :

$$1 \xrightarrow{[\![C:A]\!]} T[\![A]\!] \xrightarrow{T[\![K:A^\perp]\!]} TT[\![\mathsf{N}]\!] \xrightarrow{\mu} T[\![\mathsf{N}]\!]$$

Les clôtures $\langle P, E\rangle$ sont interprétées elles aussi en deux temps :

$$1 \xrightarrow{[\![E:\Gamma]\!]} T[\![\Gamma]\!] \xrightarrow{T[\![\Gamma \vdash P:A]\!]} TT[\![A]\!] \xrightarrow{\mu} T[\![A]\!]$$

Les objets pour les types sont

$$[\![x_1 : A_1 \dots x_n : A_n \vdash x_i : A_i]\!] = \mathsf{tick}_{A_i} \circ \boldsymbol{\pi}_i$$

$$[\![\Gamma \vdash (M)N : B]\!] = \mathsf{tick}^2_{\mathsf{N}} \bullet \mathsf{ev} \bullet \langle [\![\Gamma \vdash M : A \to B]\!], [\![\Gamma \vdash N : A]\!] \rangle$$

$$[\![\Gamma \vdash \mathsf{rec}\ f\ x = M : A \to B]\!] = \bigsqcup \varphi_n$$

$$\text{où} \begin{cases} \varphi_0 = \bot \bullet ! \\ \varphi_{n+1} = (\boldsymbol{\Lambda}[\![\Gamma, f : A \to B, x : A \vdash M : B]\!]) \bullet \langle \mathsf{id}_\Gamma, \varphi_n \rangle) \end{cases}$$

$$[\![\Gamma \vdash m : \mathsf{N}]\!] = m_0 \bullet !$$

$$[\![\Gamma \vdash \mathsf{succ}\ M : \mathsf{N}]\!] = \mathsf{tick}^2_{\mathsf{N}} \bullet (\eta \circ \underline{\mathsf{succ}}) \bullet [\![\Gamma \vdash M : \mathsf{N}]\!]$$

$$[\![\Gamma \vdash \mathsf{pred}\ M : \mathsf{N}]\!] = \mathsf{tick}^2_{\mathsf{N}} \bullet (\eta \circ \underline{\mathsf{pred}}) \bullet [\![\Gamma \vdash M : \mathsf{N}]\!]$$

$$[\![\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi} : \mathsf{N}]\!] =$$

$$\mathsf{tick}^2_{\mathsf{N}} \bullet (\eta \circ \underline{\mathsf{if}}) \bullet \langle [\![\Gamma \vdash M : \mathsf{N}]\!], \langle [\![\Gamma \vdash P : \mathsf{N}]\!], [\![\Gamma \vdash Q : \mathsf{N}]\!] \rangle \rangle$$

Fig. 4.3 – Dénotations des termes

- $[\![\mathsf{N}]\!] = \mathsf{N}_\perp$
- $[\![A \to B]\!] = [\![A]\!] \Rightarrow [\![B]\!]$
- $[\![x_1 : A_1 \dots x_n : A_n]\!] = [\![A_1]\!] \times \dots \times [\![A_n]\!]$

et on étend tick à tous les types A par induction :

$$\mathsf{tick}_{A \to B} = [\![A]\!] \Rightarrow \mathsf{tick}_B = \boldsymbol{\Lambda}(\mathsf{tick}_B \bullet \mathbf{ev}_{A,B})$$

Les dénotations des termes sont en fait les dénotations des traductions dans PCF+tick : les définitions sont données en figure 4.3. (les morphismes $\underline{\mathsf{if}}, \underline{\mathsf{succ}}, \underline{\mathsf{pred}}$ sont les dénotations des constructions $\mathsf{if}\ [-]\ \mathsf{then}\ [-]\ \mathsf{else}\ [-]\ \mathsf{fi}, \mathsf{succ}\ [-]$ et $\mathsf{pred}\ [-]$ dans le modèle d'origine.)

Les environnements sont interprétés par le produit de la catégorie de Kleisli :

$$[\![\{x_1 \to C_1 \dots x_n \to C_n\} : (x_1 : A_1 \dots x_n : A_n)]\!] = \langle [\![C_1 : A_1]\!] \dots [\![C_n : A_n]\!] \rangle$$

La définition de la sémantique des continuations est sans surprises, elle est donnée en figure 4.4.

$[\![\epsilon : \mathsf{N}^\perp]\!]^K = \mathsf{id}_\mathsf{N}$

$[\![([-]C) \cdot K : (A \to B)^\perp]\!]^K = [\![K : B^\perp]\!]^K \bullet \mathsf{tick}_B \bullet (\{-\}[\![C : A]\!])$

$[\![\mathsf{succ}\ [-] \cdot K : \mathsf{N}^\perp]\!]^K = [\![K : \mathsf{N}^\perp]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet (\eta \circ \underline{\mathsf{succ}})$

$[\![\mathsf{pred}\ [-] \cdot K : \mathsf{N}^\perp]\!]^K = [\![K : \mathsf{N}^\perp]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet (\eta \circ \underline{\mathsf{pred}})$

$[\![(\mathsf{if}\ [-]\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}, E) \cdot K : \mathsf{N}^\perp]\!]^K$

$= [\![K : \mathsf{N}^\perp]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet (\eta \circ \underline{\mathsf{if}}) \bullet \langle \mathsf{id}_\mathsf{N}, \langle [\![\Gamma \vdash P : \mathsf{N}]\!], [\![\Gamma \vdash Q : \mathsf{N}]\!] \rangle \bullet [\![E : \Gamma]\!] \bullet ! \rangle$

en notant $(\{-\}c) = \mathbf{ev} \bullet \langle \mathsf{id}, c \bullet ! \rangle$:

Fig. 4.4 – Dénotations des continuations

Correction

On a défini la sémantique pour "coller" à la réduction à petit pas. La clé des preuves est que tick commute avec toutes les constructions des continuations.

Lemme 4.2.9

Pour tout $c : 1 \to T[\![A]\!]$, $\{-\}c \bullet \mathsf{tick}_{A\to B} = \mathsf{tick}_B \bullet \{-\}c$.

Démonstration : On commence par noter que, grâce aux propriétés de $\mathbf{V}$, et par définition de $\mathsf{tick}_{A\to B}$:

$$\mathbf{ev} \bullet \langle \mathsf{tick}_{A\to B} \circ \boldsymbol{\pi}_1, \mathsf{id}_{[\![A]\!]} \circ \boldsymbol{\pi}_2 \rangle = \mathbf{V}(\mathsf{tick}_{A\to B}) = \mathsf{tick}_B \bullet \mathbf{ev}$$

ce qui nous donne

$$\begin{array}{lll}
(\{-\}c) \bullet \mathsf{tick}_{A\to B} & & \\
\quad = \mathbf{ev} \bullet \langle \mathsf{id}, c \bullet ! \rangle \bullet \mathsf{tick}_{A\to B} & & \\
\quad = \mathbf{ev} \bullet (\mathsf{tick}_{A\to B} \times \mathsf{id}_{[\![A]\!]}) \bullet \langle \mathsf{id}, c \bullet ! \rangle & \text{propriétés du produit} \\
\quad = \mathsf{tick}_B \bullet \mathbf{ev} \bullet \langle \mathsf{id}, c \bullet ! \rangle & \text{comme ci-dessus} \\
\quad = \mathsf{tick}_B \bullet (\{-\}c) & \blacksquare
\end{array}$$

Lemme 4.2.10

Pour toute continuation $K : A^\perp$, $[\![K : A^\perp]\!]^K \bullet \mathsf{tick}_A = \mathsf{tick}_\mathsf{N} \bullet [\![K : A^\perp]\!]^K$.

Démonstration : On raisonne par induction sur K. La continuation vide étant interprétée par l'identité, il n'y a rien à prouver. Le cas de l'application est une application immédiate du lemme précédent :

$$\begin{aligned}
&[\![([-])C \cdot K : (A \to B)^\perp]\!]^K \bullet \mathsf{tick}_{A\to B} \\
&\quad = [\![K : B^\perp]\!]^K \bullet \mathsf{tick}_B \bullet (\{-\}[\![C : A]\!]) \bullet \mathsf{tick}_{A\to B} \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![K : B^\perp]\!]^K \bullet (\{-\}[\![C : A]\!]) \bullet \mathsf{tick}_{A\to B} && \text{hypothèse d'induction} \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![K : B^\perp]\!]^K \bullet \mathsf{tick}_B \bullet (\{-\}[\![C : A]\!]) && \text{lemme 4.2.9} \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![([-])C \cdot K : (A \to B)^\perp]\!]^K
\end{aligned}$$

Pour les autres cas, il est utile de remarquer que, si $f \in \mathcal{C}[\mathsf{N}_\perp, \mathsf{N}_\perp]$, alors

$$(\eta \circ f) \bullet \mathsf{tick}_\mathsf{N} = \mathsf{tick}_\mathsf{N} \bullet (\eta \circ f)$$

ce qui va nous permettre de prouver la propriété pour $\mathsf{succ}\ [-]$, $\mathsf{pred}\ [-]$ et $\mathsf{if}\ [-]\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}$. Les preuves sont similaires :

$$\begin{aligned}
[\![\mathsf{succ}\ [-] \cdot K : \mathsf{N}^\perp]\!]^K \circ \mathsf{tick}_\mathsf{N} &= [\![K : \mathsf{N}^\perp]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet (\eta \circ \underline{\mathsf{succ}}) \bullet \mathsf{tick}_\mathsf{N} \\
&= \mathsf{tick}_\mathsf{N} \bullet [\![K : \mathsf{N}^\perp]\!]^K \bullet (\eta \circ \underline{\mathsf{succ}}) \bullet \mathsf{tick}_\mathsf{N} \\
&= \mathsf{tick}_\mathsf{N} \bullet [\![K : \mathsf{N}^\perp]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet (\eta \circ \underline{\mathsf{succ}}) \\
&= \mathsf{tick}_\mathsf{N} \bullet [\![\mathsf{succ}\ [-] \cdot K : \mathsf{N}^\perp]\!]^K
\end{aligned}$$

■

On peut donc prouver le lemme de base de la preuve de correction :

Lemme 4.2.11

> Si $\langle C, K\rangle \rightsquigarrow \langle C', K'\rangle$, alors $[\![K]\!]^K \bullet [\![C]\!] = \mathsf{tick}_\mathsf{N} \bullet [\![K']\!]^K \bullet [\![C']\!]$.

Démonstration : Le cas de la règle pour les variables :

$$\begin{aligned}
&[\![K : A^\perp]\!]^K \bullet [\![\Gamma \vdash x_i : A_i]\!] \bullet [\![E : \Gamma]\!] \\
&\quad = [\![K : A^\perp]\!]^K \bullet \mathsf{tick}_A \bullet \boldsymbol{\pi}_i \bullet [\![E : \Gamma]\!] \\
&\quad = [\![K : A^\perp]\!]^K \bullet \mathsf{tick}_A \bullet [\![E(x_i) : A_i]\!] \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![K : A^\perp]\!]^K \bullet [\![E(x_i) : A_i]\!]
\end{aligned}$$

Les autres cas se divisent en deux classes : soit la règle "pousse" un contexte sur K, soit elle en enlève un. On traite le cas de l'application (on omet les types pour la lisibilité) :

$$\begin{aligned}
&[\![K]\!]^K \bullet [\![(M)N]\!] \bullet [\![E]\!] \\
&\quad = [\![K]\!]^K \bullet \mathsf{tick}_\mathsf{N}^2 \bullet \mathsf{ev} \bullet \langle [\![M]\!], [\![N]\!]\rangle \bullet [\![E]\!] \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![K]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet \mathsf{ev} \bullet \langle [\![M]\!], [\![N]\!]\rangle \bullet [\![E]\!] \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![K]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet \mathsf{ev} \bullet \langle [\![M]\!] \bullet [\![E]\!], [\![(N, E)]\!]\rangle \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![K]\!]^K \bullet \mathsf{tick}_\mathsf{N} \bullet \mathsf{ev} \bullet \langle \mathsf{id}, [\![(N, E)]\!] \bullet \mathsf{!}\rangle \bullet [\![M]\!] \bullet [\![E]\!] \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet [\![(N, E) \cdot K]\!]^K \bullet [\![M]\!] \bullet [\![E]\!]
\end{aligned}$$

Des deux tick, un reste dans la nouvelle continuation, et l'autre sort. La règle correspondante "consomme" le tick qui restait dans la continuation :

$$\begin{aligned}&[\![(N,E)\cdot K]\!]^K \bullet [\![\mathsf{rec}\ f\ x = M]\!] \bullet [\![E']\!]\\ &\quad = [\![K]\!]^K \bullet \mathsf{tick}_{\mathsf{N}} \bullet \mathsf{ev} \bullet \langle \mathsf{id}, [\![(N,E)]\!] \bullet !\rangle \bullet [\![\mathsf{rec}\ f\ x = M]\!] \bullet [\![E']\!]\\ &\quad = \mathsf{tick}_{\mathsf{N}} \bullet [\![K]\!]^K \bullet \mathsf{ev} \bullet \langle \mathsf{id}, [\![(N,E)]\!] \bullet !\rangle \bullet [\![\mathsf{rec}\ f\ x = M]\!] \bullet [\![E']\!]\end{aligned}$$

Il ne reste plus qu'à prouver que ce qui reste après le tick correspond bien au nouvel état de la machine. En posant

$$\begin{cases} \varphi_0 = \bot \bullet ! \\ \varphi_{n+1} = \mathbf{\Lambda}([\![\Gamma, f : A \to B, x : A \vdash M : B]\!] \bullet \langle \mathsf{id}_\Gamma, \varphi_n\rangle) \end{cases}$$

on a

$$\begin{aligned}[\![\mathsf{rec}\ f\ x = M]\!] \bullet [\![E']\!] &= \bigsqcup \varphi_n \bullet [\![E']\!]\\ &= \bigsqcup \varphi_{n+1} \bullet [\![E']\!]\\ &= \bigsqcup \mathbf{\Lambda}[\![M]\!] \bullet \langle \mathsf{id}_\Gamma, \varphi_n\rangle \bullet [\![E']\!]\\ &= \bigsqcup \mathbf{\Lambda}[\![M]\!] \bullet \langle [\![E']\!], \varphi_n \bullet [\![E']\!]\rangle\\ &= \mathbf{\Lambda}[\![M]\!] \bullet \langle [\![E']\!], \bigsqcup \varphi_n \bullet [\![E']\!]\rangle\\ &= \mathbf{\Lambda}[\![M]\!] \bullet \langle [\![E']\!], [\![(\mathsf{rec}\ f\ x = M, E')]\!]\rangle\end{aligned}$$

ce qui nous donne

$$\begin{aligned}&[\![K]\!]^K \bullet \mathsf{ev} \bullet \langle \mathsf{id}, [\![(N,E)]\!] \bullet !\rangle \bullet [\![\mathsf{rec}\ f\ x = M]\!] \bullet [\![E']\!]\\ &\quad = [\![K]\!]^K \bullet \mathsf{ev} \bullet \langle \mathsf{id}, [\![(N,E)]\!] \bullet !\rangle \bullet \mathbf{\Lambda}[\![M]\!] \bullet \langle [\![E']\!], [\![(\mathsf{rec}\ f\ x = M, E')]\!]\rangle\\ &\quad = [\![K]\!]^K \bullet \mathsf{ev} \bullet \langle \mathbf{\Lambda}[\![M]\!], [\![(N,E)]\!] \bullet !\rangle \bullet \langle [\![E']\!], [\![(\mathsf{rec}\ f\ x = M, E')]\!]\rangle\\ &\quad = [\![K]\!]^K \bullet [\![M]\!] \bullet \langle \mathsf{id}, [\![(N,E)]\!] \bullet !\rangle \bullet \langle [\![E']\!], [\![(\mathsf{rec}\ f\ x = M, E')]\!]\rangle\\ &\quad = [\![K]\!]^K \bullet [\![M]\!] \bullet \langle \langle [\![E']\!], [\![(\mathsf{rec}\ f\ x = M, E')]\!]\rangle, [\![(N,E)]\!]\rangle\\ &\quad = [\![K]\!]^K \bullet [\![M]\!] \bullet [\![E'[f \to (\mathsf{rec}\ f\ x = M, E'), x \to (N,E)]]\!]\end{aligned}$$ ■

Théorème 4.2.12 (Correction)

Si $\langle C, K\rangle \Downarrow^n m$, alors $[\![K : A^\perp]\!]^K \bullet [\![C : A]\!] = m_n$

Démonstration : Par induction sur n. Si $n = 0$, alors $C = (m, E)$ pour un certain E et $K = \epsilon$. On a bien $\mathsf{id}_{[\![\mathsf{N}]\!]} \bullet m_0 \bullet ! \bullet [\![E]\!] = m_0$. Si $n > 0$, on applique le lemme précédent et l'hypothèse d'induction. ■

Adéquation

Pour prouver la réciproque, on définit des relations logiques hétérogènes :

$$\begin{aligned}\mathcal{R}_A &\subseteq \{C \mid C : A\} \times \mathcal{C}[1, T[\![A]\!]]\\ \mathcal{R}_{A^\perp} &\subseteq \{K \mid K : A^\perp\} \times \mathcal{C}[[\![A]\!], T\mathsf{N}]\\ \mathcal{R}_\Gamma &\subseteq \{E \mid E : \Gamma\} \times \mathcal{C}[1, T[\![\Gamma]\!]]\\ \mathcal{R}_{\Gamma \vdash A} A &\subseteq \{M \mid \Gamma \vdash M : A\} \times \mathcal{C}[[\![\Gamma]\!], T[\![A]\!]]\end{aligned}$$

La brique de base est la relation Σ entre états et morphismes de $1 \to T[\![\mathsf{N}]\!]$:

$$\Sigma = \{(\langle C, K\rangle, f) \mid f = m_n \Longrightarrow \langle C, K\rangle \Downarrow^n m\}$$

On a mis une implication pour que $\bot$ soit relié à tous les états : ce sera la cas de base de la preuve pour le cas de la récursion. Les relations entre continuations sont définies par induction sur le type :

$$\begin{array}{rcl} \mathcal{R}_{\mathsf{N}^\bot} & = & \{(K,k) \mid \forall n, (\langle n, \emptyset, K\rangle, k \bullet n_0) \in \Sigma \wedge k \circ \bot = \bot\} \\ \mathcal{R}_{(A\to B)^\bot} & = & \{([-]C \cdot K, k \bullet \{-\}c) \mid (C,c) \in \mathcal{R}_A \wedge (K,k) \in \mathcal{R}_{B^\bot}\} \end{array}$$

où $\{-\}c = \mathsf{tick}_B \bullet \mathbf{ev} \bullet \langle \mathsf{id}, c \bullet !\rangle$. Les relations entre clôtures sont définies par dualité :

$$\mathcal{R}_A = \{(C,c) \mid \forall (K,k) \in \mathcal{R}_{A^\bot}, (\langle C, K\rangle, k \bullet c) \in \Sigma\}$$

On étend ces définitions aux environnements et aux termes ouverts :

$$\begin{array}{rcl} \mathcal{R}_{x_1:A_1 \ldots x_n:A_n} & = & \{(E,e) \mid \forall i, (E(x_i), \boldsymbol{\pi}_i \bullet e) \in \mathcal{R}_{A_i}\} \\ \mathcal{R}_{\Gamma\vdash A} & = & \{(M,f) \mid \forall (E,e) \in \mathcal{R}_\Gamma, ((M,E), f \bullet e) \in \mathcal{R}_A\} \end{array}$$

Lemme 4.2.13

- Si $(K,k) \in \mathcal{R}_{\mathsf{N}^\bot}$, alors

$$(\mathsf{succ}\ [-] \cdot K,\quad k \bullet \mathsf{tick}_{\mathsf{N}} \bullet (\eta \circ \underline{\mathsf{succ}})) \in \mathcal{R}_{\mathsf{N}^\bot}$$

$$(\mathsf{pred}\ [-] \cdot K,\quad k \bullet \mathsf{tick}_{\mathsf{N}} \bullet (\eta \circ \underline{\mathsf{pred}})) \in \mathcal{R}_{\mathsf{N}^\bot}$$

- Si $(K,k) \in \mathcal{R}_{\mathsf{N}^\bot}$, $(E,e) \in \mathcal{R}_\Gamma$, $(P,p) \in \mathcal{R}_{\mathsf{N}}$ et $(Q,q) \in \mathcal{R}_{\mathsf{N}}$, alors

$$(\mathsf{if}\ [-]\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi} \cdot K,\quad k \bullet \mathsf{tick}_{\mathsf{N}} \bullet (\eta \circ \underline{\mathsf{if}}) \bullet \langle \mathsf{id}_{\mathsf{N}}, \langle p, q\rangle \bullet e \bullet !\rangle) \in \mathcal{R}_{\mathsf{N}^\bot}$$

Lemme 4.2.14

Si $(K,k) \in \mathcal{R}_{A^\bot}$, alors $\mathsf{tick}_{\mathsf{N}} \bullet k = k \bullet \mathsf{tick}_A$.

Démonstration : Si $A = \mathsf{N}$, c'est la condition requise sur $\mathsf{tick}_{\mathsf{N}}$, si $A = B \to C$, on applique le lemme 4.2.9. ■

Lemme 4.2.15

Si $\langle C, K\rangle \rightsquigarrow \langle C', K'\rangle$ et $(\langle C', K'\rangle, x) \in \Sigma$ alors $(\langle C, K\rangle, \mathsf{tick}_{\mathsf{N}} \bullet x) \in \Sigma$.

Démonstration : Supposons que $\mathsf{tick} \bullet x = m_n$. On a $n > 0$ et $x = m_{n-1}$, donc $\langle C', K'\rangle \Downarrow^{n-1} m$. On obtient bien $\langle C, K\rangle \Downarrow^n m$. ■

Lemme 4.2.16 (Lemme fondamental)

- Si $\Gamma \vdash M : A$, alors $(M, [\![\Gamma \vdash M : A]\!]) \in \mathcal{R}_{\Gamma\vdash A}$,
- Si $E : \Gamma$, alors $(E, [\![E : \Gamma]\!]) \in \mathcal{R}_\Gamma$,
- Si $K : A^\perp$, alors $(K, [\![K : A^\perp]\!]^K) \in \mathcal{R}_{A^\perp}$.

Démonstration : Pour les termes, on procède par induction sur la preuve de $\Gamma \vdash M : A$. On se donne $(E, e) \in \mathcal{R}_\Gamma$ et $(K, k) \in \mathcal{R}_{A^\perp}$, et il faut prouver

$$(\langle (M, E), K\rangle, k \bullet [\![\Gamma \vdash M : A_i]\!] \bullet e) \in \Sigma$$

Cas $x_1 : A_1 \dots x_n : A_n \vdash x_i : A_i$. On a $[\![\Gamma \vdash x_i : A]\!] \bullet e = \mathsf{tick}_{A_i} \bullet \boldsymbol{\pi}_i \bullet e$. On a, pour tout $(K, k) \in \mathcal{R}_{A_i^\perp}$, $\langle (x_i, E), K\rangle \rightsquigarrow \langle E(x_i), K\rangle$. Par hypothèse $(E(x_i), \boldsymbol{\pi}_i \circ e) \in \mathcal{R}_{A_i}$, donc $(\langle E(x_i), K\rangle, k \bullet \boldsymbol{\pi}_i \bullet e) \in \Sigma$. Par le lemme 4.2.15, $(\langle (x_i, E), K\rangle, \mathsf{tick}_\mathsf{N} \bullet k \bullet \boldsymbol{\pi}_i \bullet e) \in \Sigma$. Le lemme 4.2.9 nous donne $\mathsf{tick}_\mathsf{N} \bullet k \bullet \boldsymbol{\pi}_i \bullet e = k \bullet \mathsf{tick}_{A_i} \bullet \boldsymbol{\pi}_i \bullet e$, ce qui permet de conclure.

Le cas de la récursion est le plus intéressant. On suppose $M = \mathsf{rec}\, f\, x = P$ et $\Gamma, f : A \to B, x : A \vdash P : B$. On a $K = ([-]C) \cdot K'$ et $k = k' \bullet \mathsf{tick}_B \bullet (\{-\}c)$ pour certains $(C, c) \in \mathcal{R}_A$ et $(K', k') \in \mathcal{R}_B$. On prouve que $(M, \varphi_n) \in \mathcal{R}_{\Gamma\vdash A\to B}$. Pour φ_0, c'est évident car k est strict (par définition de $\mathcal{R}_{A^\perp}$), donc $k \bullet \perp \bullet e = \perp$ ne peut pas être un m_n.

$$\begin{aligned}
&k \bullet \varphi_{n+1} \bullet e \\
&\quad = k' \bullet \mathsf{tick}_B \bullet (\{-\}c) \bullet \boldsymbol{\Lambda}[\![P]\!] \bullet \langle \mathsf{id}, \varphi_n\rangle \bullet e \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet k' \bullet (\{-\}c) \bullet \boldsymbol{\Lambda}[\![P]\!] \bullet \langle \mathsf{id}, \varphi_n\rangle \bullet e \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet k' \bullet (\{-\}c) \bullet \boldsymbol{\Lambda}[\![P]\!] \bullet \langle e, \varphi_n \bullet e\rangle \\
&\quad = \mathsf{tick}_\mathsf{N} \bullet k' \bullet [\![P]\!] \bullet \langle \langle e, \varphi_n \bullet e\rangle, c\rangle
\end{aligned}$$

On a, par hypothèse $(E, e) \in \mathcal{R}_\Gamma$ et $(C, c) \in \mathcal{R}_A$, et par hypothèse de récurrence $((M, E), \varphi_n \bullet e) \in \mathcal{R}_{A\to B}$, donc

$$(E[f \to (M, E), x \to C], \bullet\langle \langle e, \varphi_n \bullet e\rangle, c\rangle) \in \mathcal{R}_{\Gamma, f:A\to B, x:A}$$

Comme, par hypothèse d'induction, $(P, [\![P]\!]) \in \mathcal{R}_{\Gamma, f:A\to B, x:A\vdash A\to B}$ et puisque $(K', k') \in \mathcal{R}_{B^\perp}$, on a bien

$$(\langle P, E[f \to (M, E), x \to C], K\rangle, k' \bullet [\![P]\!] \bullet \langle \langle e, \varphi_n \bullet e\rangle, c\rangle) \in \Sigma$$

Par le lemme 4.2.15, on a bien

$$(\langle M, E, K\rangle, k \bullet \varphi_{n+1} \bullet e) \in \Sigma$$

On a prouvé que, pour tout n, $(M, \varphi_n) \in \mathcal{R}_{\Gamma\vdash A\to B}$. Il est facile de prouver que la relation est close par limites : on obtient bien $(M, \bigsqcup \varphi_n) \in \mathcal{R}_{\Gamma\vdash A\to B}$, comme attendu.

Pour les autres cas, on utilise le lemme 4.2.15 et l'hypothèse d'induction sur les sous-termes. On traitera seulement le cas de l'application, on a $M = (P)Q$ avec $\Gamma \vdash P : B \to A$ et $\Gamma \vdash Q : B$. On a, d'une part,

$$\langle((P)Q, E), K\rangle \rightsquigarrow \langle(P, E), ([-](Q, E)) \cdot K\rangle$$

et d'autre part

$$\begin{aligned} & k \bullet [\![(P)Q : A]\!] \bullet e \\ & \quad = k \bullet \mathsf{tick}_A^2 \bullet (\{-\}([\![Q]\!] \bullet e)) \bullet [\![P]\!] \bullet e \\ & \quad = \mathsf{tick}_{\mathsf{N}} \bullet k \bullet \mathsf{tick}_A \bullet (\{-\}([\![Q]\!] \bullet e)) \bullet [\![P]\!] \bullet e \end{aligned}$$

Par le lemme 4.2.15, il suffit de prouver que

$$(\langle(P, E), ([-](Q, E)) \cdot K\rangle, k \bullet \mathsf{tick}_A \bullet (\{-\}([\![Q]\!] \bullet e)) \bullet [\![P]\!] \bullet e) \in \Sigma$$

Par hypothèse d'induction, $(Q, [\![Q]\!]) \in \mathcal{R}_B$, donc

$$(([-](Q, E)) \cdot K, k \bullet \mathsf{tick}_A \bullet (\{-\}([\![Q]\!] \bullet e))) \in \mathcal{R}_{B \to A}$$

Encore par hypothèse d'induction, $(P, [\![P]\!]) \in \mathcal{R}_{B \to A}$, et on conclut en rappelant que $(E, e) \in \mathcal{R}_\Gamma$.

Pour les environnements et les clôtures, on raisonne par induction sur la dérivation de typage.

Pour les continuations, on procède par induction sur la preuve de $K : A^\perp$. On se donne $(C, c) \in \mathcal{R}_A$.

Si $K = \epsilon$, alors $A = \mathsf{N}$, alors pour tout n, $[\![\epsilon]\!]^K \bullet n_0 = n_0$ et $\langle n, \emptyset, \epsilon\rangle \Downarrow^0 n$.

Si $K = ([-]C) \cdot K'$, alors $A = B \to C$ et $[\![([-]C) \cdot K' : (B \to C)^\perp]\!]^K = [\![K' : C^\perp]\!]^K \bullet \mathsf{tick}_B \bullet (\{-\}[\![C : A]\!])$. On conclut en rappelant que $(C, [\![C : A]\!]) \in \mathcal{R}_A$.

Si $K = (\mathsf{succ}\ [-]) \cdot K'$, alors $A = \mathsf{N}$. Pour tout n, $[\![K]\!]^K \bullet n_0 = [\![K']\!]^K \bullet \mathsf{tick}_{\mathsf{N}} \bullet (\eta \circ \underline{\mathsf{succ}}) \bullet n_0 = \mathsf{tick}_{\mathsf{N}} \bullet [\![K']\!]^K \bullet n + 1_0$. D'autre part, $\langle n, \emptyset, K\rangle \rightsquigarrow \langle n+1, \emptyset, K'\rangle$. Par hypothèse d'induction, $(\langle n+1, \emptyset, K'\rangle, [\![K']\!]^K \bullet n + 1_0) \in \Sigma$, donc $(\langle n, \emptyset, K\rangle, [\![K]\!]^K \bullet n_0) \in \Sigma$ également. Les autres cas sont similaires. ■

Théorème 4.2.17 (Adéquation)

Si $[\![\langle C, K\rangle]\!] = m_n$, alors $\langle C, K\rangle \Downarrow^n m$.

Démonstration : Appliquer le lemme fondamental. ■

La comonade $T \times -$

On a isolé les conditions sur la catégorie et la monade pour obtenir un modèle correct et adéquat. Dans cette partie, on va présenter une monade particulière, qui rend automatiquement la catégorie de Kleisli cartésienne fermée.

L'intuition est simple : il s'agit de voir le temps comme une ressource consommée par le programme. Il est donc naturel d'ajouter un paramètre τ aux dénotations, qui sera interrogé à chaque tic. Par exemple, en supposant que τ est entier, on pourrait définir $\mathsf{tick}; M$ par $\mathsf{if}\ \tau\ \mathsf{then}\ M\ \mathsf{else}\ M\ \mathsf{fi}$. On définit une nouvelle traduction de PCF dans PCF :

- $\langle x \rangle = \lambda\tau : T.\mathsf{tick}; x$,
- $\langle (M)N \rangle = \lambda\tau : T.\mathsf{tick}; \mathsf{tick}; (\langle M \rangle \tau)(\langle N \rangle \tau)$,
- etc.

Catégoriquement, cela correspond à la monade $T \Rightarrow -$. Pour simplifier les définitions, on va plutôt utiliser une comonade, $T \times -$: il s'agit en fait d'isoler une variable libre τ qu'on ajoute au contexte. Nous ne prétendons évidemment pas qu'on peut en général changer une monade en comonade. Ici, comme nous sommes dans une catégorie cartésienne fermée, la correspondance est immédiate. Toujours avec $\mathsf{tick}; M = \mathsf{if}\ \tau\ \mathsf{then}\ M\ \mathsf{else}\ M\ \mathsf{fi}$, la traduction devient :

- $\langle x \rangle = \mathsf{tick}; x$,
- $\langle (M)N \rangle = \mathsf{tick}; \mathsf{tick}; (\langle M \rangle)\langle N \rangle$,
- etc.

ce qui est beaucoup plus simple.

On rappelle la composition dans la catégorie de co-Kleisli de $f : T \times A \to B$ et de $g : T \times B \to C$:

$$T \times A \xrightarrow{\mu} T \times (T \times A) \xrightarrow{Tf} T \times B \xrightarrow{g} C$$

La force tensorielle $t_{A,B} : T \times (A \times B) \to (T \times A) \times B$ est tout simplement l'isomorphisme d'associativité du produit. Les opérations $\mathbf{\Lambda}$ et $\mathbf{V}$ sont obtenues à partir de Λ et V en réordonnant le contexte :

- $\eta = \underline{\lambda}\langle \tau, a \rangle.a$,
- $\mu = \underline{\lambda}\langle \tau, a \rangle.\langle \tau, \langle \tau, a \rangle \rangle$,
- $t = \underline{\lambda}\langle \tau, \langle a, b \rangle \rangle.\langle \langle \tau, a \rangle, b \rangle$,
- $\mathbf{\Lambda} f = \underline{\lambda}\langle \tau, b \rangle.\underline{\lambda}a.f\langle \tau, \langle a, b \rangle \rangle$,
- $\mathbf{V} f = \underline{\lambda}\langle \tau, \langle a, b \rangle \rangle.f\langle \tau, b \rangle a$.

Il est alors facile de vérifier que :

Proposition 4.2.18

$\langle T \times -, \eta, \mu, t \rangle$ est une comonade forte. $- \times -$ définit un produit cartésien

> pour la catégorie de co-Kleisli et $\mathbf{\Lambda}, \mathbf{V}$ définissent une adjonction entre $A \times -$ et $A \Rightarrow -$.

Il ne reste plus alors qu'à définir :
- l'objet T,
- les morphismes $m_n : T \to \mathsf{N}\bot$,
- le morphisme $\mathsf{tick}_\mathsf{N} : T \times \mathsf{N}_\bot \to \mathsf{N}_\bot$

et à vérifier que :
- pour tout morphisme $f : T \times \mathsf{N}_\bot \to \mathsf{N}_\bot$, $\mathsf{tick}_\mathsf{N} \bullet f = f \bullet \mathsf{tick}_\mathsf{N}$,
- pour tout $f : T \to \mathsf{N}_\bot$, $\mathsf{tick}_\mathsf{N} \bullet f = m_n$ si et seulement si $n > 0$ et $f = m_{n-1}$.

On va présenter deux modèles construits sur ce principe : un modèle de domaines traditionnel, en utilisant les espaces de cohérence avec les multicliques (pour pouvoir compter les occurrences), et un modèle de jeux, où les tics sont demandés l'un après l'autre avant de fournir le résulat final.

4.3 Modèles temporisés

On a vu qu'on ne pouvait pas utiliser les catégories de fonctions (car le nombre d'utilisations d'un argument n'est pas représenté), on va donc se tourner vers des catégories qui ne sont pas bien pointées, où on peut "compter" : les tics seront représentés dans un espace très simple, avec un seul point : on récupèrera le n de m_n en comptant le nombre d'occurrences de cet élément. C'est ce qu'on se propose de faire dans la suite, d'abord dans les espaces de cohérence, puis dans les jeux.

4.3.1 Modèle d'espaces de cohérence

Cohérence

On commence par rappeler les définitions des espaces de cohérences (lire [Gir95] et [EB97] pour plus de détails).

Un espace de cohérence $X = (|X|, \Xi_X)$ est un ensemble X équipé d'une relation symétrique et réflexive Ξ_X (la cohérence). La cohérence stricte Π_X est définie par $x \; \Pi_X \; x' \iff x \; \Xi \; x' \wedge x \neq x'$.

Une clique de l'espace de cohérence X est un sous-ensemble s de $|X|$, tel que $\forall x, x' \in s, x \; \Xi_X \; x'$. Une multiclique est un multiensemble m de X, tel que l'ensemble de ses éléments, noté $|m|$, est une clique de X. On note $\uplus$ l'union de multiensembles.

L'espace de cohérence produit $X \times Y$ est défini par $|X \times Y| = |X| + |Y|$ (union disjointe) et deux éléments sont cohérents si l'un est dans S et l'autre

dans Y, ou bien si les deux sont dans X (resp. Y) et ils sont cohérents dans X (resp. Y). La propriété recherchée est : une multiclique de $X \times Y$ est une paire d'une multiclique de X et d'une multiclique de Y.

L'espace pour la flèche linéaire $X \multimap Y$ est défini par $|X \multimap Y| = |X| \times |Y|$ et $(x, y)\ \Xi_{X \multimap Y}\ (x', y')$ si

$$\begin{cases} x\ \Xi_X\ x' \implies y\ \Xi_Y\ y' \\ x\ \Pi_X\ x' \implies y\ \Pi_Y\ y' \end{cases}$$

L'espace pour l'exponentielle $!X$ est défini comme l'ensemble des multicliques finies de X, deux multicliques étant cohérentes si leur union est une multiclique.

La catégorie de co-Kleisli de la comonade ! est cartésienne fermée, et la flèche intuitionniste $X \Rightarrow Y$ est, par définition, $!X \multimap Y$. Pour clarifier des définitions, on explicite l'espace de cohérence $X \Rightarrow Y$: $|X \Rightarrow Y|$ est l'ensemble des paires d'une multiclique de X et d'un élément de Y, et$(m, y)\ \Xi_{X \Rightarrow Y}\ (m', y')$ si, quand $m \uplus m'$ est une multiclique alors $y\ \Xi_Y\ y'$, et quand $m \uplus m'$ est une multiclique et que $m \neq m'$ alors $y\ \Pi\ y'$.

Ainsi, la composée d'une clique $f : X \Rightarrow Y$ et d'une clique $g : Y \Rightarrow Z$ est l'ensemble des $(m_X, z) \in |X \Rightarrow Z|$ tels que :

- il existe $m_Y = \{y_1, \ldots, y_n\}$ tel que $(m_Y, z) \in g$,
- il existe $m_X^1 \ldots m_X^n$ tels que $\forall i, (m_X^i, y_i) \in f$ et $m_X = m_X^1 \uplus \cdots \uplus m_X^n$.

Une instance de la comonade $T \times -$

Pour l'objet T, on choisit naturellement l'espace de cohérence à un point $|T| = \{*\}$: chaque tic va "consommer" un $*$ dans T. Les multicliques finies de T sont les multiensembles $*^n = \{*, \ldots, *\}$ avec n occurrences de $*$, pour chaque $n \in N$.

Les définitions des transformations monadiques nous donnent dans ce cas :

$$\begin{array}{rcl} \eta_A & = & \{(\{a\}, (\emptyset, a)) \mid a \in |A|\} \\ \mu_A & = & \{(\{(m_1, (m_2, a))\}, (m_1 \uplus m_2, a)) \mid m_1, m_2 \in |!T|, a \in |A|\} \\ \mathbf{\Lambda} f & = & \{(g, (t, (a, b))) \mid ((g, a), (t, b)) \in f\} \\ \mathbf{V} f & = & \{((g, a), (t, b)) \mid (g, (t, (a, b))) \in f\} \end{array}$$

On voit bien que μ additionne les nombres de chaque T.

On montre comment calculer l'étoile de Kleisli $f^* = Tf \circ \mu$. Elle est l'ensemble des cliques engendrées par ces deux sortes d'éléments :

$$\begin{array}{ccccccccccccc} T & \times & B & \xrightarrow{\mu} & T & \times & (T & \times & B) & \xrightarrow{Tf} & T & \times & C \\ *^n & & \beta & & & & *^n & & \beta & & & & c \\ *^1 & & & & *^1 & & & & & & * & & \end{array}$$

où $((*^n, \beta), c\}$ est dans f et $n \in N$. On voit bien comment μ propage les tics des deux T : ceux qui viennent de f ($*^n$ sur la figure) et ceux qui viennent du contexte ($*^1$ sur la figure).

On définit $\mathsf{tick}_{\mathsf{N}} = \{(*^1, \{m\}), m)\}$: il faut une étape pour propager la valeur m. Cette construction ne marche qu'au type de base, c'est pourquoi nous avons défini tick aux types d'ordre supérieur.

Par exemple, soit x le calcul qui demande 3 étapes pour renvoyer 0. et f un morphisme de T dans $N_\perp \Rightarrow N_\perp$ qui utilise deux fois son argument avant de renvoyer 1 en 5 étapes.

$$\begin{array}{ccc} T & \xrightarrow{x} & N_\perp \\ *^3 & & 0 \end{array} \qquad\qquad \begin{array}{ccccc} T & \xrightarrow{f} & N_\perp & \Rightarrow & N_\perp \\ *^4 & & \{0,0\} & & 1 \end{array}$$

En les composant, on voit apparaître, comme attendu, 1 en $10 = 4 + 3 + 3$ étapes :

$$\begin{array}{ccccccccc} T & \xrightarrow{\mu} & T & \times & T & \xrightarrow{T\times x} & T & \times & N_\perp \xrightarrow{f} N_\perp \\ *^1 & & *^1 & & & & \{* & & \\ \uplus *^1 & & \uplus *^1 & & & & * & & \\ \uplus *^1 & & \uplus *^1 & & & & * & & 1 \\ \uplus *^1 & & \uplus *^1 & & & & *\} & & \\ \uplus *^3 & & & & *^3 & & & & \{0 \\ \uplus *^3 & & & & \uplus *^3 & & & & 0\} \end{array}$$

4.3.2 Modèle de jeux

On reprend cette idée en l'appliquant au modèle de jeux. Il faut noter que Ghica [Ghi05] a construit indépendamment un modèle de coût en sémantique des jeux assez similaire au nôtre : les tics sont modélisés par des actions spéciales (des jetons) qui ne sont pas des coups, et qui ne sont pas cachés par la composition. Notre monade permet de représenter ces jetons par des coups, et de ne pas modifier la définition de la composition. Une comparaison plus détaillée suit la présentation du modèle.

On notera $(\!|\Gamma \vdash M : A|\!)$ la dénotation de $\Gamma \vdash M : A$ dans le modèle de jeux traditionnel (sans tics).

On applique la construction $T \times -$ à la catégorie des arènes et des stratégies innocentes et bien parenthésées. L'objet T sera l'arène à deux

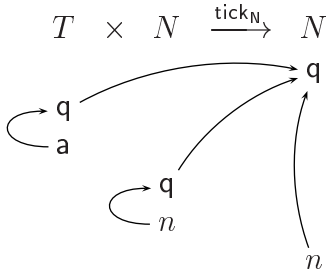
points Σ. Les seules stratégies de Σ sont $\bot = \{\epsilon\}$ et $* = \{\epsilon, \mathsf{qa}\}$, c'est bien un espace à un point.

Les stratégies de $T \to N$ sont donc de trois types :

- $\bot_n$, dont la partie maximale est $\mathsf{q}_N(\mathsf{q}_T A_T)^n$, la stratégie qui s'arrête au bout de n tics,
- $\bot_\omega = \bigsqcup \bot_k$, la stratégie qui demande toujours plus de temps mais qui ne répond pas,
- m_n, dont la partie maximale est $\mathsf{q}_N(\mathsf{q}_T A_T)^n m_N$, la stratégie qui répond m au bout de n tics.

On pourra remarquer ques contrairement à la sémantique traditionnelle, le calcul infini $\bot_\omega$ n'est pas identifié avec les calculs qui s'arrêtent $\bot_k$.

tick_N est la stratégie qui demande un tic, puis la valeur de son argument, et renvoie cette valeur. Plus formellement, elle est définie par les parties maximales suivantes, pour $n \in N$:



Il est facile de vérifier que :

- $\mathsf{tick} \bullet \bot_n = \bot_{n+1}$,
- $\mathsf{tick} \bullet \bot_\omega = \bot_\omega$,
- $\mathsf{tick} \bullet m_n = m_{n+1}$.

Comme tick est injective, on a bien $\mathsf{tick} \circ f = m_n$ si et seulement si $n > 0$ et $f = m_{n-1}$.

Pour illustrer la propagation des tics par composition, on considère le programme $\mathsf{succ\ succ}\ x$. On rappelle que $[\mathsf{succ\ succ}\ x] = \mathsf{tick}^2; \mathsf{succ}\ (\mathsf{tick}^2; \mathsf{succ}\ x)$. L'interaction est montrée en figure 4.5. En projetant sur les arènes externes,

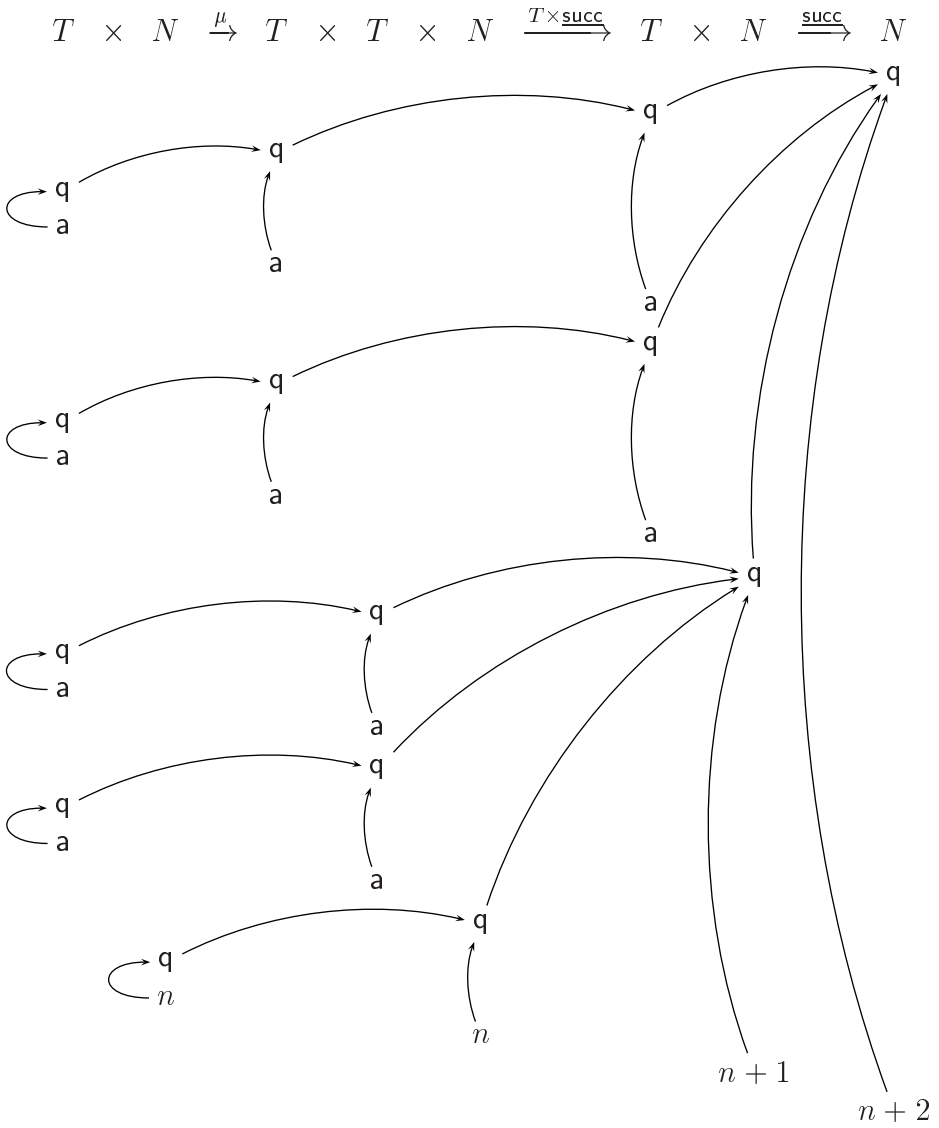
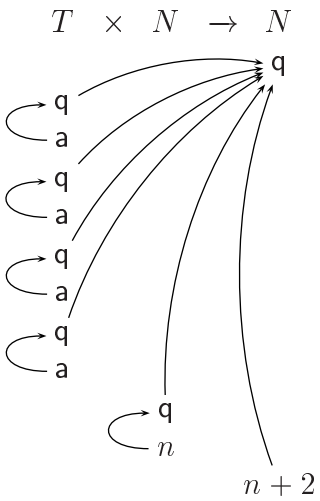


Fig. 4.5 – Exemple d'interaction

on obtient



Définition 4.3.1 (Amélioration sémantique)

Soient σ, τ des stratégies de $T \to A$. On dit que σ améliore τ, ce qu'on note $\sigma \lesssim \tau$ si pour toute statégie $\alpha : T \times A \to N$,

$$\alpha \bullet \tau = m_n \implies \exists n' \leq n, \alpha \bullet \sigma = m_{n'}$$

Il faut remarquer que cette définition va dans le sens contraire du préordre observationnel : τ est plus définie que σ.

Complète adéquation

Définition 4.3.2 (Projection)

Si $\sigma : T \times A \to B$ est une stratégie, on définit sa projection $\mathsf{GetVal}(\sigma) : A \to B$ par

$$\mathsf{GetVal}(\sigma) = \sigma \circ \langle \mathsf{skip}, \mathsf{id}_A \rangle$$

où $\mathsf{skip} : A \to T$ est la stratégie de vue maximale $\mathsf{q}_T\mathsf{a}_T$. En fait, on fournit des tics à la stratégie, et on ne garde que les coups "intéressants".

Proposition 4.3.3

Pour tout $\sigma : T \times A \to B$, $\mathsf{GetVal}(\sigma) = \{s\restriction_{A,B} \mid s \in \sigma\}$. De plus, si I est une interaction entre $T \times \sigma$ et τ :

$$T \times (T \times A) \xrightarrow{T \times \sigma} T \times B \xrightarrow{\tau} C$$

alors $I\restriction_{A,B,C}$ est une interaction entre $\mathsf{GetVal}(\sigma)$ et $\mathsf{GetVal}(\tau)$.

Lemme 4.3.4

1. $\mathsf{GetVal}(\mathsf{tick}) = \mathsf{id}_N$,
2. $\mathsf{GetVal}(\sigma \bullet \tau) = \mathsf{GetVal}(\sigma) \circ \mathsf{GetVal}(\tau)$,
3. $\mathsf{GetVal}(\mathbf{\Lambda}\sigma) = \Lambda\mathsf{GetVal}(\sigma)$,
4. $\mathsf{GetVal}(\mathbf{V}\sigma) = V\mathsf{GetVal}(\sigma)$,

Démonstration : 1. Évident.

2.
$$\begin{array}{rcl}
\mathsf{GetVal}(\sigma \bullet \tau) & = & \sigma \circ T \times \tau \circ \mu \circ \langle \mathsf{skip}_A, \mathsf{id}_A \rangle \\
& = & \sigma \circ T \times \tau \circ \langle \mathsf{skip}_A, \langle \mathsf{skip}_A, \mathsf{id}_A \rangle \rangle \\
& = & \sigma \circ \langle \mathsf{skip}_A, \tau \circ \langle \mathsf{skip}_A, \mathsf{id}_A \rangle \rangle \\
& = & \sigma \circ \langle \mathsf{skip}_A, \mathsf{GetVal}(\tau) \rangle \\
& = & \sigma \circ \langle \mathsf{skip}_B, \mathsf{id}_B \rangle \circ \mathsf{GetVal}(\tau) \\
& = & \mathsf{GetVal}(\sigma) \circ \mathsf{GetVal}(\tau)
\end{array}$$

3. On vérifie facilement que $\langle \mathsf{skip}_{A\times B}, \mathsf{id}_{A\times B} \rangle = \alpha \circ \mathsf{id}_A \times \langle \mathsf{skip}_B, \mathsf{id}_B \rangle$. On a alors
$$\begin{array}{rcl}
\mathsf{GetVal}(\mathbf{\Lambda}\sigma) & = & \mathbf{\Lambda}\sigma \circ \langle \mathsf{skip}_B, \mathsf{id}_B \rangle \\
& = & \Lambda(\sigma \circ \alpha) \circ \langle \mathsf{skip}_B, \mathsf{id}_B \rangle \\
& = & \Lambda(\sigma \circ \alpha \circ \mathsf{id}_A \times \langle \mathsf{skip}_B, \mathsf{id}_B \rangle) \\
& = & \Lambda(\sigma \circ \langle \mathsf{skip}_{A\times B}, \mathsf{id}_{A\times B} \rangle) \\
& = & \Lambda(\mathsf{GetVal}(\sigma))
\end{array}$$

Démonstration : similaire au cas précédent. ■

Proposition 4.3.5

Si $\Gamma \vdash M : A$, alors $\mathsf{GetVal}(\llbracket \Gamma \vdash M : A \rrbracket) = (\!| \Gamma \vdash M : A |\!)$.

Démonstration : Par induction sur la dérivation de $\Gamma \vdash M : A$, en appliquant le lemme précédent. ■

Les résultats suivants sont dus à Ghica. Les preuves sont des décalques de celles de [Ghi05].

Lemme 4.3.6

Soient σ, τ deux stratégies de $T \Rightarrow A$. Si $\sigma \not\lesssim \tau$, alors soit $\mathsf{GetVal}(\sigma) \not\geq \mathsf{GetVal}(\tau)$, soit il existe des parties $s \in \sigma$, $t \in \tau$ telles que :
- $s\restriction_A = t\restriction_A$,
- $s\restriction_T = (\mathsf{qa})^{n_s}$ et $t\restriction_T = (\mathsf{qa})^{n_t}$ avec $n_s < n_t$.

Démonstration : On suppose que $\sigma \not\lesssim \tau$ et que $\mathsf{GetVal}(\sigma) \supseteq \mathsf{GetVal}(\tau)$. Soit $\alpha : T \times A \to N$ tel que $\alpha \bullet \tau = m_{n_\tau}$ et $\alpha \bullet \sigma \neq m_{n'}$ pour tout $n' \leq n_\tau$.

Si $\alpha \bullet \sigma = \bot_k$ pour un certain k ou si $\alpha \bullet \sigma = \bot_\omega$, alors $\mathsf{GetVal}(\alpha) \circ \mathsf{GetVal}(\sigma) = \mathsf{GetVal}(\alpha \bullet \sigma) = \bot$. Comme $\mathsf{GetVal}(\alpha) \circ \mathsf{GetVal}(\tau) = \mathsf{GetVal}(\alpha \bullet \tau) = m$, on a (par monotonie de la composition) $\mathsf{GetVal}(\sigma) \not\supseteq \mathsf{GetVal}(\tau)$, ce qui contredit l'hypothèse.

Ainsi, il existe $n_\sigma > n_\tau$ tel que $\alpha \bullet \sigma = m_{n_\sigma}$. Soient I_σ et I_τ les interactions de α avec respectivement σ et τ dans la catégorie de Kleisli :

$$T_{12} \xrightarrow{\mu} T_1 \times T_2 \xrightarrow{T_1 \times \sigma} T_1 \times A \xrightarrow{\alpha} N$$

Par le lemme 4.3.3, $I_\sigma \restriction_{A,B}$ est une interaction entre $\mathsf{GetVal}(\sigma)$ et $\mathsf{GetVal}(\alpha)$ (resp. τ). Comme l'interaction est complète et $\mathsf{GetVal}(\sigma) \supseteq \mathsf{GetVal}(\tau)$, $s\restriction_A = I_\sigma\restriction_A = I_\tau\restriction_A = t\restriction_A$.

Par définition de μ, les coups dans T_{12} sont des copies de T_1 and T_2, et par hypothèse $I_\sigma \restriction_{T_{12}} = (\mathsf{qa})^{n_\sigma}$ (resp. I_τ, n_τ). Par alternance entre joueur et opposant, comme l'interaction est complète, il y a un nombre pair de coups dans chaque arène. $I_\sigma \restriction_{T_1}$ est égal à $I_\tau \restriction_{T_1}$ (ils ne dépendent que de α), mettons $(\mathsf{qa})^{n_\alpha}$. En calculant la différence, on obtient $s\restriction_{T_2} = (\mathsf{qa})^{(n_\sigma - n_\alpha)}$ (resp. t, n_τ). On conclut en rappelant que $n_\sigma > n_\tau$. ■

Théorème 4.3.7 (Complète adéquation)

Le modèle de jeux de PCF temporisé est complètement adéquat, c'est-à-dire que $\Gamma \vdash P \lesssim Q : A$ si et seulement si $[\![\Gamma \vdash P : A]\!] \lesssim [\![\Gamma \vdash Q : A]\!]$.

Démonstration : On commence par prouver l'implication de gauche à droite (correction) On suppose que $[\![P]\!] \lesssim [\![Q]\!]$. Si $C[]$ est un contexte tel que $C[Q] \Downarrow^n m$, alors par correction de la sémantique, $[\![C[Q]]\!] = m^n$. Il est facile de vérifier que, pour $X = P, Q$, $[\![C[X]]\!] = [\![C]\!] \bullet [\![X]\!]$. Par définition de l'amélioration sémantique, $[\![C[P]]\!] = m_{n'}$ pour un $n' \leq n$, et par adéquation de la sémantique, $C[P] \Downarrow^{n'} m$.

Prouvons maintenant la réciproque. On suppose que $[\![P]\!] \not\lesssim [\![Q]\!]$. On applique le lemme 4.3.6 à $\sigma = [\![P]\!]$ et $\tau = [\![Q]\!]$. Si $\mathsf{GetVal}(\sigma) \not\supseteq \mathsf{GetVal}(\tau)$, alors par le lemme 4.3.4, $(\!|P|\!) \not\supseteq (\!|Q|\!)$. Par le lemme 1.5.16, $P \not\supseteq Q$, et donc $P \not\lesssim Q$.

Dans le cas contraire, il existe $s \in \sigma, t \in \tau$ tels que :
- $s\restriction_A = t\restriction_A$,
- $s\restriction_T = (\mathsf{qa})^{n_s}$ et $t\restriction_T = (\mathsf{qa})^{n_t}$ avec $n_s > n_t$.

On écrit $u = s\restriction_A = t\restriction_A$, une partie de A. Soit $u' = \mathsf{q}u0$, une partie de $A \Rightarrow \mathsf{N}$. Comme les stratégies compactes sont définissables, il existe un terme $C : A \to \mathsf{N}$ tel que $(\!|C|\!)$ soit la stratégie v engendrée par u'. Soit $\alpha = [\![C]\!]$. Par le lemme 4.3.4, $\mathsf{GetVal}(\alpha \circ v) = \mathsf{GetVal}(\alpha) \circ v = (\!|C|\!) \circ v = \{\epsilon, \mathsf{q}0\}$: il existe donc n_α tel que $\alpha \circ v = 0_{n_\alpha}$. Il est immédiat de vérifier

que $[\![C[P]]\!] = \alpha \bullet \sigma = 0_{n_\sigma + n_\alpha}$, ce qui nous donne, par adéquation de la sémantique, $C[P] \Downarrow^{n_\sigma + n_\alpha} 0$ (resp. Q, τ, n_τ). Comme $n_\sigma > n_\tau$, $P \not\lesssim Q$. ■

Comparaison avec les travaux de Ghica

Ghica [Ghi05] a construit indépendamment un modèle de jeux qui exprime le coût d'un programme. Sa construction repose sur deux points :

- les stratégies sont des ensembles de suites de coups de l'arène et de jetons (tokens),
- les jetons ne sont pas cachés par composition (puisqu'ils ne sont pas des coups)

C'est exactement ce que nous avons modélisé par une monade : la composition de Kleisli vise justement à propager les coups dans l'arène T.

Il nous semble plus intéressant de ne pas modifier le cadre théorique : en faisant apparaître le temps comme une monade, on peut réutiliser à l'identique les définitions de base des jeux. Les parties sont indiscutablement beaucoup plus longues dans notre modèle, mais c'est, comme souvent, le prix à payer pour une monade.

D'autre part, et ceci est peut-être encore plus important, en faisant des tics des coups, on conserve l'intuition claire d'une suite de questions et de réponses. Dans le travail de Ghica, les jetons sont nécessaires lors de la composition, mais ils ne sont pas à proprement parler ajoutés par l'une ou l'autre des stratégies : si la définition formelle des interactions est presque inchangée, l'intuition "itérative" des stratégies qui ajoutent les coups les uns après les autres est perdue. Au contraire, dans notre modèle, il est bien clair que le terme peut demander une ressource (un tic) que le contexte doit lui accorder pour que la partie continue.

Les différences restent malgré tout très formelles : la preuve de complète adéquation vue au dessus est une simple traduction de celle de Ghica. L'étude qui suit, au contraire, est sans doute impossible si on utilise des jetons.

Si on supprime les conditions d'innocence et de bon parenthésage, notre monade permet de représenter des opérateurs de timeout (la condition d'innocence interdit de compter les tics) ou de parallélisme dans le style "trampoline" de Ganz Friedman et Wand [GFW99]. Pour plus de détails, on pourra se reporter à [Lep04b].

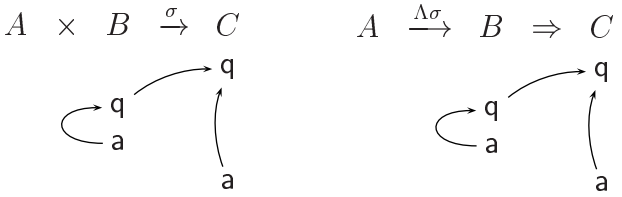
Stratégies semi-alternantes

La nature séquentielle des jeux amène assez naturellement à les considérer comme une sorte de machine abstraite : le temps de calcul serait alors

la longueur de la partie. Le problème est que la composition cache des coups : ainsi, en composant deux longues stratégies, on peut obtenir une stratégie très rapide. C'est nécessaire, car on veut que les identités restent des identités. On pourrait essayer de définir une notion de "forme canonique" pour les interactions, où les identités n'apparaîtraient pas. Cela semble très acrobatique, surtout que d'autres stratégies (comme les projections) devraient elles-aussi en être exclues. On pourrait aussi décider de se passer des identités : elles n'auraient aucun effet sur le contenu des parties, mais les allongeraient. On ne pourrait plus utiliser les catégories cartésiennes fermées, ni même les catégories ! Dans les deux cas, il apparaît que compter directement les coups est une affaire très délicate.

On cherche donc une manière indirecte de compter les coups. Ce qu'on veut éviter, c'est de cacher beaucoup de coups par composition : si on peut deviner combien ont été cachés en regardant la partie finale, on aura une idée assez précise du nombre total. Il nous faut donc une contrainte sur les stratégies qui interdise les suites de coups dans l'arène centrale lors des interactions, tout en autorisant les copycats. La question devient donc : qu'est-ce qui rend les stratégies copycats spéciales ? La clé est qu'elles alternent : à un coup à gauche, elles répondent par un coup à droite, et à un coup à droite par un coup à gauche. De plus, dans une interaction entre stratégies alternantes, il n'y a jamais deux coups consécutifs dans l'arène centrale.

Le problème est que les stratégies alternantes ne forment pas une catégorie cartésienne fermée : en renommant les coups, on brise l'alternance. Par exemple, lors d'une curryfication, on fait passer une arène de gauche à droite :



Il va donc falloir ajouter des coups de l'autre côté pour rétablir l'alternance. Ceci soulève deux problèmes : est-il possible de jouer de l'autre côté ? Comment faire pour ne pas changer le "sens" du programme ? La comonade $T \times -$ permet de résoudre les deux problèmes à la fois, pour le côté gauche : il est toujours possible pour le joueur de poser une question dans T, et cela ne change pas la projection $\mathsf{val} -$. Pour l'autre côté, on pourrait utiliser la monade duale $T \Rightarrow -$. En fait, cela n'est pas nécessaire : pour conserver la propriété qu'il n'y a jamais deux coups consécutifs dans l'arène centrale, il suffit qu'une seule des deux stratégies l'interdise. On va donc exiger des stratégies qu'elles alternent à moitié, c'est-à-dire qu'elles ne jouent jamais deux fois de suite dans l'arène de droite.

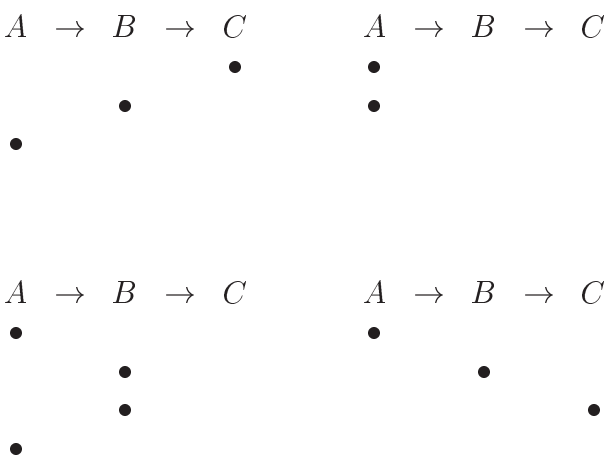
Définition 4.3.8 (Stratégie semi-alternante)

Une stratégie $\sigma : A \to B$ est semi alternante si joueur ne joue jamais dans B après un coup de l'opposant dans B.

Proposition 4.3.9

Les identités et les projections, qui sont des copycats, sont semi-alternantes. Les composées et les paires de stratégies semi-alternantes sont semi-alternantes.

Les interactions sont en fait beaucoup plus simples que dans le cas général, puisqu'il n'y a au plus que deux coups dans l'arène centrale. Les différentes suites entre deux coups dans les arènes extérieures sont



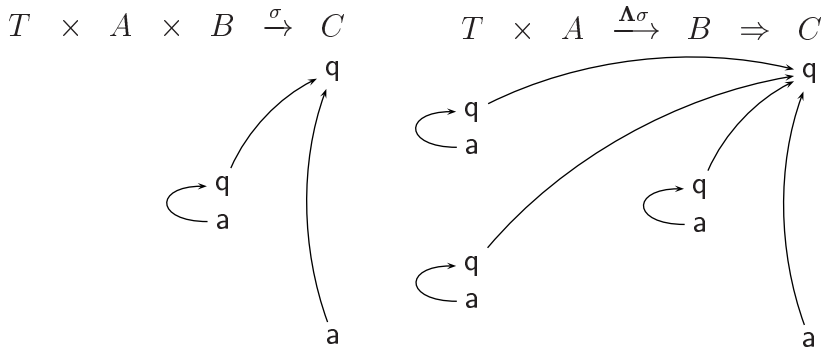
Proposition 4.3.10

Les stratégies η, μ, t sont semi-alternantes. Si σ est semi-alternante, $T \times \sigma$ aussi : la catégorie de Kleisli sur la catégorie des stratégies semi-alternantes est cartésienne.

Pour construire un modèle de PCF, il nous faut une notion de fermeture. On a vu plus haut qu'il était impossible d'utiliser la fermeture usuelle. C'est d'ailleurs souhaitable, puisque, a priori, un terme β-réduit sera plus rapide qu'un terme en forme longue. On cherche donc plutôt un procédé qui conserve encore une fois le sens (c'est-à-dire $\mathsf{val}\ -$), mais pas la longueur des parties.

Il est facile de vérifier que le morphisme d'évaluation $\mathbf{ev} = \mathbf{V}(\mathsf{id}_{A \Rightarrow B}) : A \times (A \Rightarrow B) \to B$ est semi-alternant : les seuls coups joués à droite sont des répliques de coups joués à gauche (dans B). Le problème est donc bien de construire un opérateur de curryfication. L'idée est simple : si le renommage des coups peut briser la semi-alternance, il suffit d'ajouter des coups dans

l'arène T pour la rétablir :



Plus formellement, on définit pour chaque partie s de A, une partie $\theta(s)$ de $T \times A \to A$ par :

- $\theta(s)\restriction_{A_l} = \theta(s)\restriction_{A_r}$
- $\theta(s)\restriction_T = (\mathsf{q}_T\mathsf{a}_T)^n$, ou n est la longueur de $\theta(s)\restriction_{A_r}$

$\theta(s)$ agit comme un copycat en insérant un tic entre chaque coup. On définit θ_A comme la fermeture par préfixe pair de $\{\theta(s) \mid s \in \mathcal{L}_A\}$, la stratégie engendrée par les parties sur A.

Si σ est une stratégie de $T \times A \times B \to C$, on définit $\tilde{\sigma}$, une version de σ où chaque coup dans A coûte un tic :

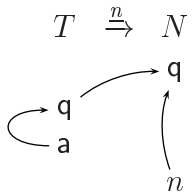
$$T\times(A\times B) \xrightarrow{\mu} T\times T\times(A\times B) \xrightarrow{T\times t} T\times(T\times A)\times B \xrightarrow{T\times(\theta_A\times B)} T\times(A\times B) \xrightarrow{\sigma} C$$

La curryfication de $\sigma : T \times (A \times B) \to C$ est $\mathbf{\Lambda}\sigma = \Lambda(\alpha \circ \tilde{\sigma})$, où α est l'isomorphisme $A \times (T \times B) \to T \times (A \times B)$.

Proposition 4.3.11

> Si σ est une stratégie semi-alternante de $T \times (A \times B) \to C$, alors $\mathbf{\Lambda}\sigma$ est une stratégie semi-alternante de $T \times B \to A \Rightarrow C$.

On peut donc définir un modèle de PCF. Le point fixe est défini de façon usuelle, par la limite d'une chaîne croissante. Les morphismes succ, pred et if sont semi-alternants, et on interprète les entiers par



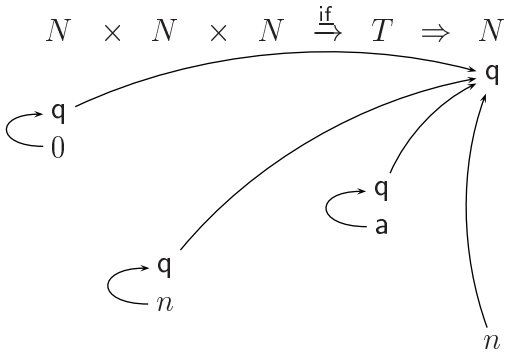
Proposition 4.3.12

> La dénotation de $M[x\backslash N]$ améliore la dénotation de $(\lambda x.M)N$.

Démonstration : Les parties de $[\![MN]\!]$ sont les mêmes que celles de $[\![M[x\backslash N]]\!]$, à ceci près qu'on a ajouté des tics autour de chaque coups de $[\![N]\!]$. On a donc $\mathsf{val}\,[\![M[x\backslash N]]\!] = \mathsf{val}\,[\![MN]\!]$, et plus de tics dans chaque partie de $[\![M[x\backslash N]]\!]$. ■

Cette idée que la dénotation d'un terme réduit est "plus petite" que celle du terme de départ rappelle le 2-λ-calcul de Hilken [Hil94] : il décrit des 2-catégories où les λ-termes sont interprétés comme des morphismes, et les réductions comme des 2-cellules. Pym et Führmann (voir par exemple [PF05]) ont également utilisé des ordres pour modéliser l'élimination des coupures.

Il peut sembler un peu curieux de représenter les entiers par une stratégie qui coûte un tic, alors que les véritables opérations (prédécesseur, successeur et test) ne coûtent rien. En fait, on pourrait tout à fait définir les stratégies semi-alternantes dualement : elles ne pourraient pas jouer deux fois de suite à gauche. Il faudrait alors utiliser la monade $T \Rightarrow -$ (au lieu de la comonade $T \times -$), et ajouter des tics à la décurryfication (au lieu de la curryfication). Les entiers pourraient être "gratuits", mais le test devrait utiliser un tic :



Les deux versions ne sont de toute façon pas réalistes : les entiers étant non bornés, les opérations ne peuvent pas être unitaires. Il faudrait plutôt, pour faire une analyse plus poussée, utiliser les entiers de Church, ou raisonner en arithmétique modulo. Rien n'empêche de toute façon de choisir pour interpréter ces opérations des stratégies qui demandent un certain nombre (fixé) de tics. Le résultat sur l'application est indépendant de ces choix.

Le modèle proposé ici n'a pas l'ambition d'offrir un cadre pour représenter fidèlement un comportement des programmes. Il faudrait de toute façon pour cela utiliser un langage plus réaliste que PCF (en particulier, en appel par valeur). Il s'agit plutôt de présenter une notion "mathématique" du temps de calcul, indépendante de la syntaxe. Outre les morphismes qui interprètent les entiers, la seule opération qui "ajoute" du temps dans le modèle est la curryfication : θ demande un tic pour chaque opération. Il semble donc que

le nombre total de tics devrait être relié au nombre d'ouvertures de variables de la machine. Néanmoins, nous n'avons pas pu prouver un résultat dans ce sens.

4.4 Conclusion

On a vu dans cette partie que la représentation d'une information sur le déroulement des programmes (le coût) n'était pas incompatible avec une sémantique compositionnelle, voire une sémantique extensionnelle (espaces de fonctions). L'utilisation d'une monade a permis de bien séparer les informations sur le temps du comportement "extensionnel" du programme, et d'éviter (grâce à la composition de Kleisli) que les dénotations puissent observer le coût de leurs arguments.

La construction est à la fois générale (on peut faire varier facilement le coût de chaque opération), et assez précise (puisqu'on peut exporter la complète adéquation du modèle de jeux traditionnel au modèle de jeux temporisé). La présence de beaucoup d'éléments non définissables (la première traduction de PCF dans PCF+tick est loin d'être surjective) ne pose ici aucun problème : comme on a bien séparé les parties observables et non observables grâce à la monade, il est facile d'appliquer une technique inspirée de la factorisation des stratégies.

On pourrait modéliser d'autres types de langages avec cette méthode :

- Le cas de l'appel par valeur est assez différent : la catégorie de Kleisli doit être seulement symétrique monoidale fermée, et surtout pas cartésienne fermée, car il faut calculer les arguments même si on ne les utilise pas. La monade la plus simple $N \times -$ convient parfaitement.
- L'appel paresseux (call-by-need) devrait rentrer aussi dans ce cadre. Il devrait suffire d'appliquer la construction aux espaces de cohérence avec l'exponentielle ensembliste traditionnelle, au lieu de l'exponentielle multiensembliste : ainsi, on ne saurait pas exactement combien de fois on a utilisé les arguments, mais on saurait s'ils ont été utilisés ou non.

Nous avons examiné plus en détail le cas de l'appel par valeur dans [Lep04a]. Ce travail n'est pas repris ici, car la définition de PCF en appel par valeur et de ses modèles est assez longue, et les résultats obtenus sont sans surprises et moins généraux.

Les stratégies semi-alternantes mériteraient sans doute une étude plus approfondie : bien que nous ne soyons pas encore parvenus à trouver une notion de réduction qui leur corresponde, on peut espérer que, peut-être moyennant quelques modifications, on puisse faire le lien avec une machine plus "concrète".

Chapitre 5

Conclusions

Sans prétention d'exhaustivité, nous avons présenté plusieurs résultats sur divers aspects de la notion d'observation. Au passage, nous avons découvert des domaines divers de la sémantique des langages de programmation : théorie des domaines, relations paramétriques, modèles catégoriques et monades, etc.

D'abord, dans le chapitre 2, nous avons présenté des travaux "traditionels" sur la complète adéquation pour PCF. Ainsi, nous avons montré que les degrés de définissabilité (dans le modèle de Scott) ne correspondent pas tous à un modèle, même pour un langage très simple. Même s'il semble a priori qu'il s'agisse d'un résultat négatif (il n'y a pas autant modèles que de degrés), il a un "corollaire" intéressant : on éliminant une fonction non définissable par quotient, on en élimine aussi beaucoup d'autres, ce qui nous donne un modèle bien meilleur qu'on aurait pu l'espérer.

Pour PCF finitaire, nous avons raffiné les hypergraphes de Bucciarelli, ce qui nous a permis d'énoncer un théorème de complétude sur la définissabilité entre fonctions sous-séquentielles. Les travaux sur la complexité montrent que c'est un problème difficile : si le résultat (décider de la définissabilité relative est NP-dur) n'est pas véritablement une surprise, nous en avons apporté une preuve, qui complète –par exemple– l'algorithme de Stoughton.

Nous avons également décrit une implémentation de la fonctionnelle H de Longley : bien que cette section n'apporte aucun nouveau résultat mathématique, ce travail nous a permis de beaucoup mieux comprendre le fonctionnement de cette fonction, et nous l'avons inclus dans cette thèse en espérant qu'elle pourra également profiter à d'autres.

Ces résultats mettent en lumière la complexité des relations entre les différentes notions d'observation étudiées : opérationnelle (contextes syntaxiques), dans le modèle (hiérarchie de modèles) et en autorisant tel ou tel comportement

(degrés de définissabilité). Rapprocher ces notions, c'est-à-dire se rapprocher de la complète adéquation, est bien un problème difficile : ces travaux permettent de mieux en comprendre certains aspects, même si nous ne l'avons pas attaqué de front.

Le chapitre suivant sur MILler applique ces techniques à un langage nettement plus ambitieux, avec des variables allouées sur le tas. Ici, nous ne visions pas la complète adéquation, mais nous voulions un cadre mathématique qui permette de prouver assez simplement des équivalences entre termes. L'utilisation des domaines de Fränkel-Mostowski (domaines FM) est, à notre connaissance, originale pour décrire l'allocation dynamique : elle simplifie très nettement la sémantique, et offre un cadre algébrique agréable car la notion de support abstrait complètement la partie de la mémoire utilisée.

Le modèle brut obtenu est assez décevant, mais des techniques traditionnelles de relations logiques paramétriques permettent de prouver beaucoup d'équivalences, et la présentation est plus simple que dans les travaux précédents [OR00, RY04], grâce à l'utilisation des domaines FM. Il reste néanmoins le problème de la réversibilité des changements (snapback) : notre modèle n'est pas à proprement parler à la pointe du progrès (il faut toutefois tenir compte de la complexité de notre langage, avec récursion et références sur des références), mais nous espérons que quelques changements mineurs dans l'esprit de [OR00, Pit97] (soulever l'ensemble des états pour interdire l'affaiblissement) permettront d'éliminer ces comportements.

Les techniques mises en œuvre dans les relations paramétriques montrent que, dans ce cas, les problèmes d'adéquation entre les notions d'observation opérationnelle et dénotationnelle sont très similaires à des problèmes de confidentialité : la partie visible de la mémoire et l'invariant correspondent respectivement aux niveaux publics et privés.

Le dernier chapitre présente nos travaux sur la représentation du coût en sémantique dénotationnelle : nous avons montré un procédé générique pour enrichir un modèle existant à l'aide d'une monade, sous des hypothèses assez légères. Nous avons présenté trois modèles : le modèle "évident" avec la monade $N_\omega \times -$ sur CPO, et le modèle d'espaces de cohérence et le modèle de jeux avec la monade $T \times -$ (où T est un singleton). Le modèle de jeux est complètement adéquat, bien qu'on ait perdu la propriété de définissabilité. Nous avons aussi présenté les stratégies semi-alternantes, qui définissent un modèle de PCF avec une notion de temps de calcul dérivée des jeux : nous n'avons malheureusement pas encore trouvé de lien précis avec une sémantique opérationnelle à petits pas.

Ainsi, l'utilisation d'une monade permet de bien séparer les différentes sortes d'observation. Le temps de calcul n'étant pas observable opérationnellement,

on aurait pu craindre que le modèle enrichi s'avère beaucoup plus mauvais que le modèle de départ. En fait, grâce à la composition de Kleisli, les contextes dénotationnels n'ont pas non plus accès au coût de l'élément : ainsi, le modèle de jeux reste complètement adéquat (la propriété de définissabilité est perdue, mais lors de la première traduction de PCF dans PCF+tick).

Certains de ces résultats méritent encore d'être approfondis. Ainsi, le modèle de MILler devrait pouvoir être amélioré, pour éliminer le snapback, et modéliser des types plus complexes (types inductifs). Les liens avec le flot d'information, dans la lignée des travaux de Sumii et Pierce [SP01], semblent également prometteurs. L'étude du coût pourrait aussi être continuée : on pourrait l'appliquer à un langage plus réaliste que PCF, en appel paresseux ou en appel par valeur, en cherchant peut-être à modéliser directement le langage plutôt qu'en passant par une traduction arbitraire qui introduit fatalement des éléments non-définissables.

Bibliographie

[AC98] Roberto Amadio et Pierre-Louis Curien : Typed lambda-calculi and domains. Cambridge University Press, 1998.

[AHM98] Samson Abramsky, Kohei Honda et Guy McCusker : A fully abstract game semantics for general references. In Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science, 1998.

[AJM00] Samson Abramsky, Radha Jagadeesan et Pasquale Malacaria : Full abstraction for PCF. Information and Computation, 163: 409–470, 2000.

[BC82] G. Berry et P. L. Curien : Sequential algorithms on concrete data structures. TCS, 20:265–321, janvier 1982.

[BCL85] Gérard Berry, Pierre-Louis Curien et Jean-Jacques Lévy : Full abstraction for sequential languages : The state of the art. In Maurice Nivat et J. C. Reynolds, éditeurs : Algebraic Methods in Semantics, pages 89–132. Cambridge University Press, 1985.

[BE93] Antonio Bucciarelli et Thomas Ehrhard : A theory of sequentiality. TCS, 113(2):273–291, 7 juin 1993.

[BE94] Antonio Bucciarelli et Thomas Ehrhard : Sequentiality in an extensional framework. Information and Computation, 110(2): 265–296, 1 mai 1994.

[Ben04] Nick Benton : Simple relational correctness proofs for static analyses and program transformations. In Proceedings of POPL'04, janvier 2004. Revised version available from http ://research.microsoft.com/ nick/publications.htm.

[Ber79] Gérard Berry : Modèles complètement adéquats et stables des λ-calculs typés. Thèse de doctorat d'état, Université de Paris 7, 1979.

[BG95] Guy E. Blelloch et John Greiner : Parallelism in sequential functional languages. In Proceedings of the Symposium on

Functional Programming and Computer Architecture, pages 226–237, juin 1995.

[BL04] Antonio Bucciarelli et Benjamin Leperchey : Hypergraphs and degrees of parallelism : a completeness result. In FOSSACS : International Conference on Foundations of Software Science and Computation Structures. LNCS, 2004.

[BL05] Nick Benton et Benjamin Leperchey : Relational reasoning in a nominal semantics for storage. In Typed Lambda-Calculus and Applications, LNCS. Springer, avril 2005.

[BLP03] Antonio Bucciarelli, Benjamin Leperchey et Vincent Padovani : Relative definability and models of unary PCF. In TLCA : Typed Lambda Calculus and Applications. LNCS, 2003.

[BM02] Antonio Bucciarelli et Pasquale Malacaria : Relative definability of boolean functions via hypergraphs. Theor. Comput. Sci., 278(1-2):91–110, 2002.

[BM04] Patrick Baillot et Virgile Mogbil : Soft lambda-calculus : A language for polynomial time computation. In FOSSACS. LNCS, 2004.

[Buc97] Antonio Bucciarelli : Degrees of parallelism in the continuous type hierarchy. Theoretical Computer Science, 177(1):59–71, 30 avril 1997.

[CE94] Loïc Colson et Thomas Ehrhard : On strong stability and higher-order sequentiality. In Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 103–108, Paris, France, 4–7 juillet 1994. IEEE Computer Society Press.

[CF92] Robert Cartwright et Matthias Felleisen : Observable sequentiality and full abstraction. In Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, pages 328–342. ACM, ACM, janvier 1992.

[Chr00] Juliusz Chroboczek : Game semantics and subtyping. In 15th Symposium on Logic in Computer Science (LICS' 00), pages 192–203. IEEE, juin 2000.

[Cur93] Pierre-Louis Curien : Categorical Combinators, Sequential Algorithms and Functional Programming. Birkhauser, revised edition édition, 1993.

[dB72] N. G. de Bruijn : Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. Indag. Math., 34:381–392, 1972.

[EB97] Thomas Ehrhrard et Nuño Barreiro : Anatomy of an extensional collapse. 1997.

[Ehr99] Thomas Ehrhard : A relative PCF-definability result for strongly stable functions and some corollaries. Information and Computation, 152, 1999.

[Esc98] Martin H. Escardó : A metric model of PCF. Laboratory for Foundations of Computer Science, University of Edinburgh. Unpublished research note, presented at the Workshop on Realizability Semantics and Applications, 1999, April 1998.

[GFW99] Steven E. Ganz, Daniel P. Friedman et Mitchell Wand : Trampolined style. In Proc. 1999 ACM SIGPLAN International Conference on Functional Programming, pages 18–27, Paris, septembre 1999.

[Ghi05] Dan R. Ghica : Slot games. In Symposium on Principles of Programming Languages, janvier 2005.

[Gir95] Jean-Yves Girard : Linear logic : its syntax and semantics. In J.-Y. Girard, Y. Lafont et L. Regnier, éditeurs : Advances in Linear Logic, pages 1–42. Cambridge University Press, 1995. London Mathematical Society Lecture Note Series 222, Proceedings of the 1993 Workshop on Linear Logic, Cornell Univesity, Ithaca.

[GJ79] Michael R. Garey et David S. Johnson : Computers and intractability. freeman and company, 1979.

[GM04] Dan R. Ghica et A. S. Murawski : Angelic Semantics of Fine-Grained Concurrency. In FOSSACS : International Conference on Foundations of Software Science and Computation Structures, LNCS. Springer, 2004.

[GMW79] M. J. Gordon, A. J. Milner et C. P. Wadsworth : Edinburgh LCF, volume 78 de Lecture Notes in Computer Science. Springer-Verlag, 1979.

[Gre97] John Greiner : Semantics-based Parallel Cost Models and Their Use in Provably Efficient Implementations. Thèse de doctorat, Carnegie Mellon University, School of Computer Science, 1997.

[Gur91] D. J. Gurr : Semantic Frameworks for Complexity. Thèse de doctorat, Department of Computer Science, University of Edinburgh, janvier 1991. Available as CST-72-91 and ECS-LFCS-91-130.

[Har99] Russ Harmer : Games and full abstraction for nondeterministic languages. Thèse de doctorat, University of London, 1999.

[Hil94] Barnaby P. Hilken : Towards a proof theory of rewriting : the simply-typed 2-λ calculus. Rapport technique UCAM-CL-TR-336, University of Cambridge, Computer Laboratory, mai 1994.

[HO00] J. Martin E. Hyland et C.-H. Luke Ong : On full abstraction for PCF : I, II, and III. Information and Computation, 163(2): 285–408, décembre 2000.

[Hof99] M. Hofmann : Linear types and non-size-increasing polynomial time computation. In 14th Symposium on Logic in Computer Science (LICS'99), pages 464–473. IEEE, 1999.

[Hug89] R. J. M. Hughes : Why functional programming matters. The Computer Journal, 2(32):98–107, avril 1989.

[Jef92] Alan Jeffrey : A linear time process algebra. In Proceedings of Computer Aided Verification (CAV '91), volume 575 de LNCS, pages 432–442. Springer, juillet 1992.

[JT93] Achim Jung et Jerzy Tiuryn : A new characterization of lambda definability. Lecture Notes in Computer Science, 664:245–257, 1993.

[Kri90] Jean-Louis Krivine : Lambda-calcul, types et modèles. Masson, 1990.

[Kri04] Jean-Louis Krivine : A call-by-name lambda calculus machine. To appear in Higher Order and Symbolic Computation, 2004.

[Lai98] Jim Laird : A semantic analysis of control. Thèse de doctorat, University of Edinburgh, 1998.

[Lai03a] Jim Laird : Bistability : an extensional characterization of sequentiality. In Proceedings of CSL'03, 2003.

[Lai03b] Jim Laird : A fully abstract bidomain model of unary FPC. In Proceedings of TLCA'03. Springer, 2003. Available at http ://www.cogs.susx.ac.uk/users/jiml/ufpc.ps.gz.

[Lai04] Jim Laird : Locally boolean domains. Submitted, 2004.

[Lep04a] Benjamin Leperchey : Call-by-name, call-by-value and time. http ://www.pps.jussieu.fr/ leperche/pub/time.pdf, octobre 2004.

[Lep04b] Benjamin Leperchey : Time and games. Prépublication PPS, référence PPS//04/02//n°27 (pp), février 2004.

[Lev02] P. B. Levy : Possible world semantics for general storage in call-by-value. In J. Bradfield, éditeur : Proceedings of the Annual

Conference of the European Association for Computer Science Logic (CSL'02), volume 2471 de Lecture Notes in Computer Science. Springer-Verlag, septembre 2002.

[Loa98] Ralph Loader : Unary PCF is decidable. Theoretical Computer Science, 206:317–329, 1998.

[Loa01] Ralph Loader : Finitary PCF is not decidable. Theoretical Computer Science, 366:341–364, 2001.

[Lon99] John Longley : When is a functional program not a functional program ? In Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99), volume 34.9 de ACM Sigplan Notices, pages 1–7, N.Y., septembre 27–29 1999. ACM Press.

[Lon02] John R. Longley : The sequentially realizable functionals. Annals of pure and applied logic, 117:1–93, 2002.

[Lon03] John R. Longley : Universal types and what they are good for. In Domain Theory, Logic and Computation : proceedings of the second Internation Symposium on Domain Theory. Kluwer academic publishers, 2003.

[Mac71] Saunders MacLane : Categories for the working mathematician. Springer-Verlag, New York, 1971.

[Mil77] Robin Milner : Fully abstract models of typed λ-calculi. Theoretical Computer Science, 4:1–22, 1977.

[Mog89] Eugenio Moggi : Computational lambda-calculus and monads. In Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asiloomar, CA, pages 14–23, 1989.

[Mog91] Eugenio Moggi : Notions of computation and monads. Information and Computation, 93:55–92, 1991.

[MS88] A. R. Meyer et K. Sieber : Towards a fully abstract semantics for local variables : Preliminary report. In Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), janvier 1988.

[MS00] Andrew Moran et David Sands : Improvement in a lazy context : An operational theory for call-by-need, mars 17 2000.

[MTHM97] R. Milner, M. Tofte, R. Harper et D. MacQueen : The Definition of Standard ML (Revised). The MIT Press, 1997.

[OR95] Peter W. O'Hearn et Jon R. Riecke : Kripke logical relations and PCF. Information and Computation, 120:107–116, 1995.

[OR00] P. W. O'Hearn et J. C. Reynolds : From Algol to polymorphic linear lambda-calculus. Journal of the ACM, 47(1):167–223, janvier 2000.

[ORY01] Peter W. O'Hearn, John C. Reynolds et Hongseok Yang : Local reasoning about programs that alter data structures. In Computer Science Logic, 2001.

[OT95] P. W. O'Hearn et R. D. Tennent : Parametricity and local variables. Journal of the Association for Computing Machinery, 42(3):658–709, May 1995.

[OT97] Peter W. O'Hearn et R. D. Tennent : ALGOL-like Languages. Progress in Theoretical Computer Science. Birkhäuser, 1997. Two volumes.

[Pao04] Luca Paolini : Lambda-theories : some investigations. Thèse de doctorat, Gênes, 2004.

[PF05] Gordon Pym et Carsten Führmann : Order-enriched categorical models of the classical sequent calculus. Journal of pure and applied algebra, 2005.

[PHA+97] John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid et Philip Wadler : Report on the programming language Haskell, version 1.4. http ://www.haskell.org/onlinereport/, avril 1997.

[Pit97] A. M. Pitts : Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn et R. D. Tennent, éditeurs : Algol-Like Languages, volume 2, chapitre 17, pages 173–193. Birkhauser, 1997.

[Pit02] Andrew M. Pitts : Operational semantics and program equivalence. In P. Dybjer G. Barthe et J. Saraiva, éditeurs : Applied Semantics, volume 2395 de Lecture Notes in Computer Science. Springer-Verlag, 2002.

[PJW93] Simon Peyton-Jones et Philip Wadler : Imperative functional programming. In Symposium on Principles of Programming Languages. ACM press, janvier 1993.

[Plo73] G. D. Plotkin : Lambda definability and logical relations. Rapport technique, Department of AI, University of Edinburgh, 1973.

[Plo77] Gordon W. Plotkin : LCF considered as a programming language. Theoretical Computer Science, 5, 1977.

[Plo80] Gordon W. Plotkin : Lambda-definability in the full type hierarchy. In To H.B. Curry : Essays on combinatory logic, lambda calculus and formalism, pages 363–373. Academic Press, 1980.

[PP02a] Gordon D. Plotkin et John Power : Erratum to notions of computations determine monads. Available from http ://homepages.inf.ed.ac.uk/gdp/publications/, 2002.

[PP02b] Gordon D. Plotkin et John Power : Notions of computations determine monads. In Foundations of software science and computations structures 2002, volume 2303 de LNCS, 2002.

[PS98] Andrew M. Pitts et Ian D. Stark : Operational Reasoning for Functions with Local State. In A. D. Gordon and A. M. Pitts, éditeur : Higher Order Operational Techniques in Semantics, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[Roy04] J. S. (James S.) Royer : On the computational complexity of Longley's H functional. Theoretical Computer Science, 318(1–2):225–241, juin 2004.

[RY04] Uday S. Reddy et Hongseok Yang : Correctness of data representations involving heap data structures. Science of Computer Programming, 50(1–3):129–160, mars 2004.

[San90a] D. Sands : Calculi for Time Analysis of Functional Programs. Thèse de doctorat, Department of Computing, Imperial College, London, septembre 1990.

[San90b] D. Sands : Complexity analysis for a higher-order language. In N. Jones, éditeur : Proceedings of 3rd European Symposium on Programming,Copenhagen, volume 432 de Lecture Notes in Computer Science, pages 361–376, mai 1990.

[San95] David Sands : A naïve time analysis and its theory of cost equivalence. Journal of Logic and Computation, 5(4), 1995.

[San98] David Sands : Improvement theory and its applications. In A. D. Gordon et A. M. Pitts, éditeurs : Higher Order Operational Techniques in Semantics, Publications of the Newton Institute, pages 275–306. Cambridge University Press, 1998.

[Saz76] V. Yu. Sazonov : Expressibility of functions in D. Scott's LCF language. Algebra i Logika, 15:308–330, 1976.

[Shi04] M. R. Shinwell : The Fresh Approach : Functional Programming with Names and Binders. Thèse de doctorat, Computer Laboratory, University of Cambridge, décembre 2004.

[Sie92] Kurt Sieber : Reasoning about sequential functions via logical relations. In Applications of categories in computer science, pages 258–269. Cambridge University Press, 1992.

[SP01] Eijiro Sumii et Benjamin C. Pierce : Logical relations for encryption. In Computer Security Foundations Workshop, juin 2001. To appear in Journal of Computer Security.

[SP05] M. R. Shinwell et A. M. Pitts : On a Monadic Semantics for Freshness. Theoretical Computer Science, 2005. To appear.

[SS71] D. S. Scott et C. Strachey : Toward a mathematical semantics for computer languages. In Proceedings of the Symposium on Computers and Automata, volume 21 de Microwave Research Institute Symposia, pages 19–46, 1971.

[SS99] Manfred Schmidt-Schauss : Decidability of behavioural equivalence in unary PCF. Theoretical Computer Science, 216: 363–373, 1999.

[Sta94] Ian. D. B. Stark : Names and higher order functions. Thèse de doctorat, Computer Laboratory, University of Cambridge, décembre 1994. Available as Technical Report 363.

[Sto94] Allen Stoughton : Mechanizing logical relations. Lecture Notes in Computer Science, 802:359–377, 1994.

[TG00] Robert D. Tennent et Dan R. Ghica : Abstract Models of Storage. Higher-Order and Symbolic Computation, 13(1/2): 119–129, 2000.

[Tra75] M. B. Trakhtenbrot : On representation of sequential and parallel functions. In Fourth Symposium on Mathematical Foundations of Computer Science, numéro 32 de LNCS. Springer, 1975.

[VS03] Kathryn Van Stone : A denotational approach to measuring complexity in functional programs. Thèse de doctorat, Carnegie Mellon University, 2003.

[WF94] A. K. Wright et Matthias Felleisen : A syntactic approach to type soundness. Information and Computation, 115(1):38–94, novembre 1994.