Réflexion, calculs et logiques

Hubert Godfroy

UNIVERSITÉ DE LORRAINE

DGA

École doctorale IAEM Lorraine

Réflexion, calculs et logiques

THÈSE

présentée et soutenue publiquement le 6 octobre 2017

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention Informatique)

par

Hubert Godfroy

Composition du jury

Rapporteurs :	Gilles Dowek (Directeur de recherche, LSV) Christian Rétoré (Professeur, Université de Montpellier)
Examinateurs :	Paul-André Melliès (Chargé de recherche, IRIF) Claudia Faggian (Chargée de recherche, IRIF) Véronique Cortier (Directrice de Recherche, CNRS)
Directeur :	Jean-Yves Marion (Professeur, Université de Lorraine)

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

Remerciements

Mon directeur Jean-Yves Marion m'a guidé pendant les quatre années de cette thèse. Et par dessus tout, il a su garder un optimisme indéfectible. Je lui en suis très reconnaissant. Je remercie également Véronique Cortier de m'avoir suivi et encouragé en m'apportant son point de vue extérieur, ainsi que Paul-André Melliès pour nos rencontres profitables. Merci enfin aux membres du jury qui ont accepté d'évaluer ce travail.

Pour leur accueil et la cordialité de nos échanges quotidiens, je tiens à saluer Emmanuel, Mathieu, Guillaume, Fabrice, Simon, Romain et Emmanuel. Grâce à Simon, Ludovic, Éric, Svyat et Pierre, la vie au laboratoire s'est vue parsemée de nombreuses rencontres agréables. Le bureau B235 aura abrité un temps de (longues) discussions avec Hugo et Aurélien, dont l'inutilité le disputait à la mauvaise foi, mais sans jamais manquer d'intérêt ni de bonne humeur ! Je salue en également Laurent, pour son aide constante, même dans les démarches administratives les plus velues, ainsi que pour ses conseils pendants ces quatre ans.

0e30d959d27570d48f09308763429a09a466eda2

Sommaire

Introduction

Virus Un *programme* est une séquence finie de commandes à exécuter dans un ordre précis. Un programme définit un *comportement*, c'est à dire un moyen de passer d'un état à un autre.

Lorsque l'on parle d'un programme, on peut considérer deux objets : soit le comportement qu'il définit, soit de l'objet représentant la séquence finie de ses commandes. Dans le premier cas, on parle de la sémantique *extensionnelle* du programme (elle répond à la question "Que fait le programme ?"), et dans le second cas de sa sémantique *intentionnelle* (elle répond à la question "Comment fonctionne le programme ?").

Dans certaines situations la sémantique intentionnelle n'a que peu d'importance. Par exemple, lorsque l'on lance un programme sur son ordinateur, sa forme ou sa taille ne comptent pas. On souhaite juste que le programme fasse ce qu'on lui demande. Dans le cas des virus informatiques, le comportement du virus est ce qui est responsable du "désagrément" (keylogger, cheval de Troie, ransomware,...). Mais dans d'autre cas, la manière de s'exécuter du programme compte. Par exemple, les programmes antivirus reconnaissent les programmes malveillants à partir de leur syntaxe (donc de manière intentionnelle).

Pour ne pas être détectés tout en conservant le même comportement, les virus informatiques doivent alors subir une transformation changeant leur sémantique intentionnelle sans toucher à leur sémantique extensionnelle. On appelle *obfuscation* cette transformation. Différentes solutions d'obfuscation existent : codes morts, prédicats opaques [CTL98], chevauchements de code [Thi15], *etc.*

Une autre de ces méthode est l'auto-modification. Un programme auto-modifiant est capable de lire et modifier son propre code pendant son exécution. Il est difficile d'analyser le comportement de ces programmes car les parties qui décrivent ce comportement (charge) sont justement celles qui sont inaccessibles avant l'exécution. En pratique, le programme est composé avant exécution d'un partie chiffrée, et l'exécution de ce programme déchiffre cette partie avant de l'exécuter. La charge est donc cachée sous plusieurs couches de chiffrements successives.

Le terme académique de l'auto-modification est la *réflexion*.

Réflexion Ce concept désigne la capacité d'un programme à "se voir" et "agir" directement sur lui-même.

D'une part, la réflexion "en lecture" donne accès à un programme à une version *réifiée* de lui même. Cette particularité de la réflexion est étudiée depuis Kleene et son Second Théorème de Récursion [Kle38]. Il montre que pour toute fonction calculable, il existe

un programme calculant cette fonction appliquée à son propre code. Un application très connue de ce théorème est l'existence de *quines* dans tout langage de programmation acceptable, c'est à dire de programmes produisant leurs propres codes. Cette idée d'auto reproduction est reprise par Von Neumann et ses *automates auto-réplicateurs* [Neu66] et plus tard Conway dans son *jeu de la vie* [Gar70]. Dans le langage C, un programme est capable d'accéder à son propre code grâce aux pointeurs de fonction.

D'autre part, la réflexion en "écriture" permet de modifier son propre code. Par exemple, en Python il est possible de demander l'exécution d'une chaine de caractères. Cette notion est directement liée à celle de l'*interprétation* qui permet d'exécuter une donnée.

Que ce soit en "lecture" ou en écriture, la réflexion peut-être résumée comme la *possibilité de voir un programme comme une donnée, et réciproquement*. Pour étudier la réflexion, un point capital sera donc l'utilisation de cette dualité, notamment en explicitant ce que sont les données et les programmes dans les langages utilisés, et comment on passe d'un état à l'autre.

Interprétation logique du calcul La correspondance de Curry-Howard assure qu'un calcul correspond à une preuve logique. Le λ-calcul est ainsi associé à la Logique Intuitionniste [How80]. Étendu par l'opérateur de capture de continuation callcc, il est associé à la logique classique [Gri90]. On est donc en droit de se demander ce que signifie un calcul réflexif *logiquement*. En particulier, à quoi correspond la dualité données/programmes, l'exécution d'une donnée, la réification d'un programme.

On soutiendra que la Logique Linéaire est un bon cadre pour l'étude des langages réflexifs.

Résumé de la thèse

Partie 1 : Cinématique de la réflexion

La dualité programmes/données, et le passage d'un état à l'autre suscite une vision très "physique" de la réflexion. La première partie sera consacrée à l'étude des "mouvements" produits par la réflexion.

Chapitre 1 Les notions élémentaires de cette thèse sont définies (outils mathématiques). On rappelle également la définition d'un langage de programmation (*langages acceptables*) suivant le modèle de Jones [Jon97 ; JGS93]. Cette définition se base sur deux notions : les programmes (ce qui est peut exécuté) et les données (ce qui peut-être lu et modifié, les entrées/sorties des programmes). Cette distinction entre ces deux mondes est la pierre angulaire de notre travail, qui consistera à étudier le rapport entre eux, comment il est possible de passer d'un monde à l'autre. Le chapitre présente également des résultats génériques sur les langages de programmations *acceptables*, notamment le Théorème de Récursion de Kleene, point fondamental de notre étude de la réflexion car liant les entrées (données) d'un programme avec sa sémantique.

Chapitre 2 Un programme réflexif est capable de donner à une donnée une sémantique *extensionnelle* (on ne regarde que les entrées et les sorties du programme qu'encode cette donnée), ou de donner à un programme une sémantique *intentionnelle* (on s'intéresse à comment fonctionne le programme). Pour la première, on transforme une donnée en un programme (réflexion) et dans l'autre on transforme un programme en une donnée (réification). On utilise ces deux principes dans l'élaboration d'une représentation des langages réflexifs dans laquelle un programme peut *interpréter* un autre, qui lui même en interprète un autre, *etc.* (tour de réflexions). Cette représentation a plusieurs applications dans la conception de programmes réflexifs.

Chapitre 3 On présente un modèle de machine abstraite explicitant les phénomènes réflexifs. Cette machine donne une représentation des réflexions et des réifications comme des mouvements de blocs de codes (registres) entre une zone où les registres sont exécutés et une zone où les registres sont lisibles et modifiables. On montre que cette machine est un modèle de calcul correct en donnant divers exemples de programmes. On exhibe notamment les programmes fondamentaux (interpréteurs, spécialiseurs) décrits dans le Chapitre 1. Enfin, on donne une mesure des mouvements des registres dans la machine, et on donne une définition d'une réflexion et d'une réification dans cette machine.

Chapitre 4 En s'intéressant plus particulièrement aux phénomènes des réflexions (exécution de données), on établit une sémantique abstraite d'un programme réflexif. L'abstraction se base sur le découpage de l'exécution réflexive en plusieurs exécutions sans réflexion.

Partie 2 : Interprétation logique

Chapitre 5 En guise d'introduction, on donne la définition du λ-calcul, et de ses principales propriétés (confluence, typage, ...). Ce chapitre répond à deux besoins : dans un premier temps, il nous sert de modèle de rédaction pour les autres parties où des langages proches du λ-calcul seront définis. D'autre part, il introduit les notions essentielles des langages fonctionnels que l'on retrouvera à chaque fois qu'on définira un nouveau langage.

Chapitre 6 On rappelle quelques bases sur la logique linéaire. Cette logique introduit les notions de gestion des ressources (que nous n'utiliserons pas ici) et l'opérateur exponentiel $!A$, ainsi que les deux règles Dereliction et Promotion pour l'introduction à gauche et à droite de l'exponentielle. Nous verrons les problèmes qui apparaissent lorsque l'on souhaite associer une syntaxe à la logique linéaire, qui conserve les propriétés de réduction du sujet et de cohérence. On montre ce que ces propriétés impliquent sur la sémantique opérationnelle de la syntaxe, et notamment sur l'élimination des coupures sur les types $!A$: seules les preuves se terminant par Promotion peuvent être "poussées" lors de la suppression des coupures. On abordera également la Soft Linear Logic, un sous-système de la logique linéaire, remplaçant la règle Promotion par une règle dérivée, la règle SoftPromotion.

Chapitre 7 On présente une version étendue du lambda calcul ayant des capacités réflexives. Pour cela notre calcul défini des termes dont l'exécution est suspendue. L'idée est que ces termes correspondent à des données. À partir de cette supposition, on construit le langage en vérifiant qu'il conserve les propriétés usuelles comme l'α-renommage, la confluence, ... On montrera comment ce langage s'inscrit dans le modèle des langages de programmation du Chapitre 1. Enfin on détaillera aussi ses capacités réflexives, en montrant ses lacunes au niveau de la réification. On met également ce langage en perspective avec d'autres langages de la littérature, ces langages étant connus dans le domaine de la *stage computation*, c'est à dire le domaine où le calcul est découpé en plusieurs étapes.

Chapitres 8 et 9 On donne deux systèmes de types à Λ_R, l'un s'appuyant sur la Logique Linéaire (PIL, Chapitre 9), et l'autre sur la Soft Linear Logic (SIL, Chapitre 8). Les termes gelés sont associés au type $!A$ en utilisant une règle Promotion (ou SoftPromotion). Les deux systèmes de types sont corrects, c'est à dire qu'ils ont la propriété de réduction du sujet. On lie ici *deux problèmes distincts* dans leur méthode de résolution : celui de la cohérence d'une syntaxe de la logique linéaire (Chapitre 6) et celui de la confluence de Λ_R (Chapitre 7). Ces deux problèmes se résolvent de la même manière : en n'éliminant les coupures sur $!A$ (autorisant les substitutions dans les termes gelés) que lorsque la preuve de $!A$ se termine par Promotion (lorsque le terme à substituer est un terme gelé).

Chapitre 10 Le système SIL est inclus dans PIL, c'est à dire que tous les termes typables dans SIL le sont dans PIL. De plus, il existe des termes non typables dans PIL qui ne le sont pas dans SIL. La fonction de digging $\mathsf{dig} : !A \to !!A$ en est un exemple. On montre même que le système SIL avec une fonction de digging permet de typer tous les termes de PIL, à la manière de Lafont [Laf02]. On montre que l'utilisation de la déréliction dans SIL correspond à une exécution de terme gelé.

Chapitre 11 Ce chapitre apporte une solution au problème de la réification. On introduit ici une stratégie d'évaluation dans Λ_R afin de n'autoriser la réduction que de termes clos. Dans ce cadre, on peut parler d'un opérateur de réification. On présente alors un opérateur $\mathcal{C}$ capturant la continuation courante, et la passant, sous forme réifiée, au programme, s'inspirant ainsi du travail de [MF93]. On exprime ce calcul dans une machine construite à partir d'une machine de Krivine. On donne également un système de types construit à partir de PIL. Ce système de types est correct au regard des règles de réduction, c'est à dire qu'il est stable par réduction.

Première partie

Cinématique de la réflexion

Chapitre 1

Éléments de théorie de la programmation

On présente dans ce chapitre introductif les outils mathématiques utilisés tout le long de la thèse. D'autre part, on définit les notions de base des langages de programmation, dans un cadre général. Ces notions accompagneront toute la thèse (données, programmes, interpréteurs, ...). Finalement on donne quelques résultats de la théorie des langages de programmation, dont le principal est le Théorème de Récursion de Kleene.

1.1 Outils mathématiques

1.1.1 Fonctions partielles

Soit A et B deux ensembles. On dit que f est une *fonction partielle* de A vers B, et on note $f : A \to_\perp B$, si f est une fonction totale de A vers $B \cup \{\perp\}$. Le *support* de f, noté $\mathrm{supp}(f)$, est défini par $\mathrm{supp}(f) = \{a \in A \,|\, f(a) \neq \perp\}$. On dira que f est incluse dans g (noté $f \subseteq g$) lorsque $\forall a \in \mathrm{supp}(f), f(a) = g(a)$. Si f et g sont deux fonctions partielles de A vers B à supports disjoints, alors $f \cup g$ est la fonction partielle

$$\begin{array}{rccll} f \cup g & : & A & \to_\perp & B \\ & & a \in \mathrm{supp}(f) & \mapsto & f(a) \\ & & a \in \mathrm{supp}(g) & \mapsto & g(a) \\ & & a & \mapsto & \perp \qquad \text{sinon} \end{array}$$

Si $b \in B \cup \{\perp\}$ et $a \in A$, on note $f[a \mapsto b]$ la fonction partielle telle que $\forall a' \neq a, f[a \mapsto b](a') = f(a')$ et $f[a \mapsto b](a) = b$.

1.1.2 Listes

Soit A un ensemble, on note A^* l'ensemble des *listes* sur A, c'est à dire l'ensemble des suites finies d'éléments de A. On note ϵ la liste vide et si a appartient à A, on note $a.u$ l'ajout de a en début de u et si u et v sont des listes sur A, alors $u \bullet v$ est la concaténation

de u et de v, c'est à dire l'opération définie inductivement par

$$\begin{cases} \epsilon \bullet v & \stackrel{\text{def}}{=} & v \\ (a.u) \bullet v & \stackrel{\text{def}}{=} & a.(u \bullet v) \quad a \in A \end{cases}$$

La liste $(a_1.(a_2.(\cdots.(a_n.\epsilon)\cdots)))$ pourra être notée $[a_1, a_2, \cdots, a_n]$. Si $\phi : A \to B$, on note $\phi^* : A^* \to B^*$ l'extension "point à point" de ϕ, c'est à dire que $\phi^*([a_1, \cdots, a_n]) \stackrel{\text{def}}{=} [\phi(a_1), \cdots, \phi(a_n)]$.

Définition 1.1.1 (Préfixe)**.** Une liste v est *préfixe* d'une liste u s'il existe une liste w telle que $u = v \bullet w$. On note $\text{pref}(u)$ l'ensemble des préfixes de u.

Définition 1.1.2 (Ensemble préfixe)**.** Un ensemble de listes E est un *ensemble préfixe* si $\forall u \in E, \text{pref}(u) \subseteq E$

1.1.3 Relations

Une *relation* sur un ensemble $E \times F$ est un élément de $\wp(E \times F)$. Si $\mathcal{R}$ est une relation de $E \times E$, on note $\mathcal{R}^*$ sa clôture réflexive transitive. On dit qu'une relation $\mathcal{R}$ est *déterministe* lorsque, pour tout a, b, c, si $a\ \mathcal{R}\ b$ et $a\ \mathcal{R}\ c$ alors $b = c$.

1.2 Langages de programmation

1.2.1 Définition

Dans cette section et les suivantes, on s'inspire de [Jon97] pour la présentation des définitions d'un langage de programmation, et des concepts qui lui sont associés.

Définition 1.2.1 (Langage de programmation)**.** Un langage de programmation $\mathcal{L}$ est le triplet $(\mathcal{P}_\mathcal{L}, \mathcal{D}_\mathcal{L}, [\![]\!]_\mathcal{L})$ où

- $\mathcal{P}_\mathcal{L}$ et $\mathcal{D}_\mathcal{L}$ sont les ensembles énumérables des programmes p et des données d,
- La fonction partielle $[\![]\!]_\mathcal{L} : \mathcal{P}_\mathcal{L} \to \mathcal{D}_\mathcal{L}^* \to \mathcal{D}_\mathcal{L}^*$ est la fonction *sémantique*, où $\mathcal{D}_\mathcal{L}^*$ est l'ensemble des listes de $\mathcal{D}_\mathcal{L}$.

On notera le fait que les fonctions calculées par les programmes prennent comme paramètre une liste de données (ce sont des fonctions d'arité variable). Ce choix est purement esthétique car il simplifie l'écriture des programmes en général.

Définition 1.2.2 (Fonction calculable)**.** On dira qu'une fonction partielle $f : \mathcal{D}_\mathcal{L}^* \to_\perp \mathcal{D}_\mathcal{L}^*$ est $\mathcal{L}$*-semi-calculable* s'il existe un programme $p_f \in \mathcal{P}_\mathcal{L}$ tel que $[\![p_f]\!]_\mathcal{L} = f$. Si en plus f est totale, alors on dira que f est $\mathcal{L}$*-calculable*.

Définition 1.2.3 (Langage complet)**.** Soient deux langages $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{D}_1, [\![]\!]_1)$ et $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{D}_2, [\![]\!]_2)$ et soit une bijection[1] $\phi : \mathcal{D}_2 \to \mathcal{D}_1$. La fonction ϕ est une fonction de codage.

1. Il existe toujours une bijection entre deux domaines $\mathcal{D}_1$ et $\mathcal{D}_2$ de deux langages, car ces ensembles sont énumérables

On dit que $\mathcal{L}_1$ est ϕ-$\mathcal{L}_2$-*complet* si pour tout programme $p_2 \in \mathcal{P}_2$, il existe un programme $p_1 \in \mathcal{P}_1$ tel que $\phi^* \circ [\![p_2]\!]_2 = [\![p_1]\!]_1 \circ \phi^*$, c'est à dire tel que le diagramme suivant commute (on rappelle que ϕ^* est l'extension "point à point" de ϕ sur les listes).

$$\begin{array}{ccc} & [\![p_2]\!]_2 & \\ \mathcal{D}_2^* & \longrightarrow & \mathcal{D}_2^* \\ \phi^* \downarrow & & \downarrow \phi^* \\ \mathcal{D}_1^* & \longrightarrow & \mathcal{D}_1^* \\ & [\![p_1]\!]_1 & \end{array}$$

On dira que $\mathcal{L}_1$ est $\mathcal{L}_2$-complet si il est **id**-$\mathcal{L}_2$-complet.

Définition 1.2.4 (Langages équivalents)**.** Soient $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{D}_1, [\![]\!]_1)$ et $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{D}_2, [\![]\!]_2)$ deux langages et $\phi : \mathcal{D}_2 \to \mathcal{D}_1$ une bijection. Le langage $\mathcal{L}_1$ est ϕ-$\mathcal{L}_2$-*équivalent* lorsque $\mathcal{L}_1$ est ϕ-$\mathcal{L}_2$-complet et $\mathcal{L}_2$ est ϕ^{-1}-$\mathcal{L}_1$-complet.

Le langage $\mathcal{L}_1$ est $\mathcal{L}_2$-équivalent lorsqu'il est **id**-$\mathcal{L}_2$-équivalent.

Propriété 1.2.5 (Symétrie)**.** *Le langage $\mathcal{L}_1$ est ϕ-$\mathcal{L}_2$-équivalent si et seulement si le langage $\mathcal{L}_2$ est ϕ^{-1}-$\mathcal{L}_1$-équivalent.*

On pourra donc dire de deux langages qu'ils sont équivalents.

Propriété 1.2.6 (Transitivité)**.** *Si $\mathcal{L}_2$ est ϕ-$\mathcal{L}_3$-complet (resp. équivalent) et $\mathcal{L}_1$ est ϕ'-$\mathcal{L}_2$-complet (resp. équivalent), alors $\mathcal{L}_1$ est $(\phi' \circ \phi)$-$\mathcal{L}_3$-complet (resp. équivalent).*

Démonstration. Posons $\mathcal{L}_i = (\mathcal{P}_i, \mathcal{D}_i, [\![]\!]_i)$ pour $i \in \{1, 2, 3\}$. Soit $p_3 \in \mathcal{P}_3$. Il existe $p_2 \in \mathcal{P}_2$ tel que $\phi'^* \circ \phi^* \circ [\![p_3]\!]_3 = \phi'^* \circ [\![p_2]\!]_2 \circ \phi^*$. Il existe aussi $p_1 \in \mathcal{P}_1$ tel que $\phi'^* \circ [\![p_2]\!]_2 \circ \phi^* = [\![p_1]\!]_1 \circ \phi'^* \circ \phi^*$. □

Dans la suite on va parler des relations que l'on peut trouver entre les langages. Si on étudie les langages $\mathcal{L}_1$ et $\mathcal{L}_2$, on adoptera couramment la notation $\mathcal{P}_1$ et $\mathcal{P}_2$ au lieu de $\mathcal{P}_{\mathcal{L}_1}$ et $\mathcal{P}_{\mathcal{L}_2}$. De même pour les autres composantes des langages.

1.2.2 Notation

On sera amené à manier différents objets. Les fonctions (objets mathématiques) seront notées en **gras**, les programmes en police sans serif, et les instructions en police `teletype`.

1.3 Langage acceptable

Suivant la définition donnée dans [Rog87], un langage est dit *acceptable* lorsqu'il est Turing-complet (Définition 1.2.4 et Section 1.3.1) et qu'il dispose d'une machine universelle (Section 1.3.3) et d'une fonction de spécialisation (Section 1.3.4). La définition complète sera donnée dans la Section 1.3.3.

1.3.1 Turing-complétude

On note $\mathcal{T} = (\mathcal{P}_\mathcal{T}, \mathcal{D}_\mathcal{T}, [\![]\!]_\mathcal{T})$ le langage où $\mathcal{P}_\mathcal{T}$ désigne l'ensemble des machines de Turing à plusieurs rubans, $\mathcal{D}_\mathcal{T}$ l'ensemble des rubans et si $p \in \mathcal{P}_\mathcal{T}$, alors $[\![p]\!]_\mathcal{T}(\mathbf{d}) = \mathbf{d}'$ lorsque la machine dont l'entrée sont les rubans de la liste $\mathbf{d}$ s'arrête sur les rubans de la liste $\mathbf{d}'$.

Définition 1.3.1 (Turing-complétude)**.** Soient $\mathcal{L} = (\mathcal{P}, \mathcal{D}, [\![]\!])$ un langage et une fonction d'encodage $\phi : \mathcal{D}_\mathcal{T} \to \mathcal{D}$. Le langage $\mathcal{L}$ est ϕ-Turing-complet lorsque $\mathcal{L}$ est ϕ-$\mathcal{T}$-complet. Le langage $\mathcal{L}$ est Turing-complet lorsqu'il est **id**-Turing-complet.

Propriété 1.3.2. *Si $\mathcal{L} = (\mathcal{P}, \mathcal{D}, [\![]\!])$ est ϕ-Turing-complet, alors il existe $\mathsf{id} \in \mathcal{P}$ et $\mathsf{dupl} \in \mathcal{P}$ tel que*

$$\begin{aligned} [\![\mathsf{id}]\!](\mathbf{d}) &= \mathbf{d} \\ [\![\mathsf{dupl}]\!](d.\mathbf{d}) &= d.d.\mathbf{d} \\ [\![\mathsf{pop}]\!](d.\mathbf{d}) &= \mathbf{d} \end{aligned}$$

Démonstration. Dans un premier temps, on note qu'il existe des machines de Turing $\mathsf{id}_\mathcal{T}$ et $\mathsf{dupl}_\mathcal{T}$ telles que $[\![\mathsf{id}_\mathcal{T}]\!]_\mathcal{T} = \mathbf{id}$ et $[\![\mathsf{dupl}_\mathcal{T}]\!]_\mathcal{T}(d_\mathcal{T}.\mathbf{d}_\mathcal{T}) = d_\mathcal{T}.d_\mathcal{T}.\mathbf{d}$ et $[\![\mathsf{pop}_\mathcal{T}]\!]_\mathcal{T}(d_\mathcal{T}.\mathbf{d}_\mathcal{T}) = \mathbf{d}$.

Il existe une bijection $\phi : \mathcal{D}_\mathcal{T} \to \mathcal{D}$ telle que $\mathcal{L}$ est ϕ-$\mathcal{T}$-complet. Donc il existe $\mathsf{id}, \mathsf{dupl} \in \mathcal{P}$ tels que $[\![\mathsf{id}_\mathcal{T}]\!]_\mathcal{T} = (\phi^*)^{-1} \circ [\![\mathsf{id}]\!] \circ \phi^*$, $[\![\mathsf{dupl}_\mathcal{T}]\!]_\mathcal{T} = (\phi^*)^{-1} \circ [\![\mathsf{dupl}]\!] \circ \phi^*$ et $[\![\mathsf{pop}_\mathcal{T}]\!]_\mathcal{T} = (\phi^*)^{-1} \circ [\![\mathsf{pop}]\!] \circ \phi^*$. Donc

$$\begin{aligned} [\![\mathsf{id}]\!] &= \phi^* \circ [\![\mathsf{id}_\mathcal{T}]\!]_\mathcal{T} \circ (\phi^*)^{-1} = \mathbf{id} \\ [\![\mathsf{dupl}]\!] &= \phi^* \circ [\![\mathsf{dupl}_\mathcal{T}]\!]_\mathcal{T} \circ (\phi^*)^{-1} \\ [\![\mathsf{pop}]\!] &= \phi^* \circ [\![\mathsf{pop}_\mathcal{T}]\!]_\mathcal{T} \circ (\phi^*)^{-1} \end{aligned}$$

Et donc

$$\begin{aligned} [\![\mathsf{dupl}]\!](d.\mathbf{d}) &= \phi^*([\![\mathsf{dupl}_\mathcal{T}]\!]_\mathcal{T}((\phi^{-1}(d)).((\phi^*)^{-1}(\mathbf{d})))) \\ &= \phi^*((\phi^{-1}(d)).(\phi^{-1}(d)).((\phi^*)^{-1}(\mathbf{d}))) \\ &= d.d.\mathbf{d} \end{aligned}$$

De même

$$[\![\mathsf{pop}]\!](d.\mathbf{d}) = \mathbf{d}$$

□

1.3.2 Domaines de calcul

Ce travail va nécessiter de passer fréquemment d'un programme à une donnée et *vice versa*. On suppose donc que dans chaque langage $\mathcal{L}$ il existe une fonction d'encodage $\mathcal{P}_\mathcal{L} \to \mathcal{D}_\mathcal{L}$. Si p est un programme, ***on notera $\overline{p}$ son encodage***.

De plus, afin de simplifier l'écriture, sauf mention explicite du contraire, on considèrera que tous les langages partagent le même ensemble de données $\mathcal{D}$.

1.3.3 Interprétation

Soient deux langages $\mathcal{L}_i = (\mathcal{P}_i, \mathcal{D}, [\![]\!]_i)$ pour $i \in \{1, 2\}$. Un *interpréteur* de $\mathcal{L}_2$ écrit dans $\mathcal{L}_1$ est un programme $\mathsf{int}_{\mathcal{L}_1}^{\mathcal{L}_2} \in \mathcal{P}_1$ (noté int_1^2 dans la suite pour plus de facilités) tel que

$$\forall p_2 \in \mathcal{P}_2, \forall d \in \mathcal{D}, \forall \mathbf{d} \in \mathcal{D}^* \quad [\![\mathsf{int}_1^2]\!]_1(\overline{p_2}.d.\mathbf{d}) = [\![p_2]\!]_2(d.\mathbf{d}).$$

Un interpréteur d'un langage $\mathcal{L}_2$ est un programme de $\mathcal{L}_1$ capable de reproduire le comportement de n'importe quel programme de $\mathcal{L}_2$, au niveau du langage $\mathcal{L}_1$. Le cas le plus simple est quand $\mathcal{L}_1 = \mathcal{L}_2$: dans cette situation, un interpréteur est appelé *auto-interpréteur* ou *machine universelle*. On le notera univ_1.

1.3.3.1 Domaines distincts

Si $\mathcal{L}_i = (\mathcal{P}_i, \mathcal{D}_i, [\![]\!]_i)$ où $\mathcal{D}_1 \neq \mathcal{D}_2$, on peut utiliser bijection $\phi : \mathcal{D}_2 \to \mathcal{D}_1$. Le programme $\mathsf{int}_1^2 \in \mathcal{P}_1$ est un interpréteur lorsque

$$\forall p_2 \in \mathcal{P}_2, \forall d_2 \in \mathcal{D}_2, \forall \mathbf{d} \in \mathcal{D}_2^* \quad ([\![\mathsf{int}_1^2]\!]_1 \circ \phi^*)(\overline{p_2}.d_2.\mathbf{d}) = (\phi^* \circ [\![p_2]\!]_2)(d_2.\mathbf{d}).$$

1.3.4 Spécialisation

Soit le langage $\mathcal{L}_2 \stackrel{\text{def}}{=} (\mathcal{P}_2, \mathcal{D}, [\![]\!]_2)$. On dit que la fonction $\mathbf{spec}^{\mathcal{L}_2} : \mathcal{P}_2 \to \mathcal{D} \to \mathcal{P}_2$ (notée dans la suite $\mathbf{spec}^2$) est une *fonction de spécialisation* si

$$\forall p \in \mathcal{P}_2, \forall d, d' \in \mathcal{D}, \forall \mathbf{d} \in \mathcal{D}^* \quad [\![\mathbf{spec}^2(p,d)]\!]_2 (d'.\mathbf{d}) = [\![p]\!]_2 (d.d'.\mathbf{d})$$

On dira que $\mathbf{spec}^2$ est implémentée dans le langage $\mathcal{L}_1$ s'il existe $\mathsf{spec}_{\mathcal{L}_1}^{\mathcal{L}_2} \in \mathcal{P}_1$ (noté spec_1^2 dans la suite) tel que

$$\forall p_2 \in \mathcal{P}_2, d \in \mathcal{D}, \mathbf{d} \in \mathcal{D}^* \quad [\![\mathsf{spec}_1^2]\!]_1 (\overline{p_2}.d.\mathbf{d}) = \overline{\mathbf{spec}^2(p_2,d)}.\mathbf{d}$$

La fonction de spécialisation est la base théorique de l'évaluation partielle, c'est à dire pré-calculer la fonction calculée par un programme p dont tous les arguments ne sont pas tous fournis. Le programme $\mathbf{spec}(p,d)$ est l'évaluation partielle de p sur son premier paramètre. On pourra se reporter à [JGS93] ou [Jon97] pour plus de détails sur le sujet.

1.3.4.1 Domaines distincts

De la même manière que dans la définition de l'interpréteur, on peut étendre la définition d'un spécialiseur dans le cas où $\mathcal{D}_1 \neq \mathcal{D}_2$ et qu'il existe une fonction injective $\phi : \mathcal{D}_2 \to \mathcal{D}_1$. Un programme $\mathsf{spec}_1^2 \in \mathcal{P}_1$ est un spécialiseur lorsque

$$\forall p_2 \in \mathcal{P}_2, d_2 \in \mathcal{D}_2, \mathbf{d} \in \mathcal{D}_2^* \quad ([\![\mathsf{spec}_1^2]\!]_1 \circ \phi^*)(\overline{p_2}.d_2.\mathbf{d}) = \phi^*(\overline{\mathbf{spec}^2(p_2,d_2)}.\mathbf{d})$$

1.3.5 Langage acceptable

Définition 1.3.3 (Langage acceptable)**.** Un langage $\mathcal{L}$ est *acceptable* lorsque

- $\mathcal{L}$ est Turing-complet,
- il existe dans les machines de Turing un interpréteur $\mathsf{int}_{\mathcal{T}}^{\mathcal{L}}$ de $\mathcal{L}$,
- il existe dans les machines de Turing un spécialiseur $\mathsf{spec}_{\mathcal{T}}^{\mathcal{L}}$ de $\mathcal{L}$, calculant la fonction de spécialisation $\mathbf{spec}^{\mathcal{L}}$.

Propriété 1.3.4. *Tout langage acceptable $\mathcal{L}$ contient une machine universelle et un spécialiseur.*

Démonstration. Le langage $\mathcal{L}$ est $\mathcal{T}$-complet, et $\mathsf{int}_{\mathcal{T}}^{\mathcal{L}}$ et $\mathsf{spec}_{\mathcal{T}}^{\mathcal{L}}$ sont des machines de Turing. Donc il existe $\mathsf{univ}_{\mathcal{L}}$ et $\mathsf{spec}_{\mathcal{L}}^{\mathcal{L}}$ tels que

$$\begin{array}{rcl} [\![\mathsf{univ}_{\mathcal{L}}]\!]_{\mathcal{L}} & = & [\![\mathsf{int}_{\mathcal{T}}^{\mathcal{L}}]\!]_{\mathcal{T}} \\ [\![\mathsf{spec}_{\mathcal{L}}^{\mathcal{L}}]\!]_{\mathcal{L}} & = & [\![\mathsf{spec}_{\mathcal{T}}^{\mathcal{L}}]\!]_{\mathcal{T}} \end{array}$$

Donc

$$[\![\mathsf{univ}_{\mathcal{L}}]\!]_{\mathcal{L}}(\overline{p}.\mathbf{d}) = [\![\mathsf{int}_{\mathcal{T}}^{\mathcal{L}}]\!]_{\mathcal{T}}(\overline{p}.\mathbf{d}) = [\![p]\!]_{\mathcal{L}}(\mathbf{d})$$
$$[\![\mathsf{spec}_{\mathcal{L}}^{\mathcal{L}}]\!]_{\mathcal{L}}(\overline{p}.d.\mathbf{d}) = [\![\mathsf{spec}_{\mathcal{T}}^{\mathcal{L}}]\!]_{\mathcal{T}}(\overline{p}.d.\mathbf{d}) = \overline{\mathbf{spec}^{\mathcal{L}}(p,d)}.\mathbf{d}$$

□

Propriété 1.3.5. *Si $\mathcal{L}$ est acceptable, alors il est* Turing*-équivalent.*

Démonstration. On sait déjà que $\mathcal{L}$ est Turing-complet. Montrons que le langage $\mathcal{T}$ est $\mathcal{L}$-complet. Soit p un programme de $\mathcal{L}$. Soit $p' \stackrel{\text{def}}{=} \mathbf{spec}^{\mathcal{T}}(\mathsf{int}_{\mathcal{T}}^{\mathcal{L}}, \overline{p})$. Alors on a

$$\begin{array}{rcl} [\![p']\!]_{\mathcal{T}}(\mathbf{d}) & = & [\![\mathbf{spec}^{\mathcal{T}}(\mathsf{int}_{\mathcal{T}}^{\mathcal{L}}, \overline{p})]\!]_{\mathcal{T}}(\mathbf{d}) \\ & = & [\![\mathsf{int}_{\mathcal{T}}^{\mathcal{L}}]\!]_{\mathcal{T}}(\overline{p}.\mathbf{d}) \\ & = & [\![p]\!](\mathbf{d}) \end{array}$$

Donc $\mathcal{T}$ est $\mathcal{L}$-complet. □

Propriété 1.3.6. *Deux langages acceptables $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{D}, [\![]\!]_1)$ et $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{D}, [\![]\!]_2)$ sont équivalents et s'interprètent l'un l'autre, c'est à dire qu'il existe des interpréteurs $\mathsf{int}_1^2 \in \mathcal{P}_1$ et $\mathsf{int}_2^1 \in \mathcal{P}_2$. De même il existe des spécialiseurs $\mathsf{spec}_1^2 \in \mathcal{P}_1$ et $\mathsf{spec}_2^1 \in \mathcal{P}_2$.*

Démonstration. Ces deux langages sont équivalents avec le langage des machines de Turing. Donc, par transitivité (Propriété 1.2.6), ils sont équivalents. De plus, chacun contient une programme universel et un spécialiseur (Propriété 1.3.4) $\mathsf{univ}_1 \in \mathcal{P}_1$ et $\mathsf{univ}_2 \in \mathcal{P}_2$, $\mathsf{spec}_1^1 \in \mathcal{P}_1$ et $\mathsf{spec}_2^2 \in \mathcal{P}_2$. Comme ces deux langages sont équivalents, il existe int_1^2 et (resp. int_2^1, spec_1^2, spec_2^1) calculant la même chose que univ_2 (resp. univ_1, spec_2^2, spec_1^1). □

Propriété 1.3.7 (Composition)**.** *Soit $\mathcal{L} = (\mathcal{P}, \mathcal{D}, [\![]\!])$ un langage acceptable. Il existe un programme $\mathsf{compo} \in \mathcal{P}$ tel que $[\![\mathsf{compo}]\!](\overline{p}.\overline{q}.\mathbf{d}) = \overline{p \circ q}.\mathbf{d}$.*

Démonstration. D'une part, la composition existe dans les machines de Turing : si p et q sont deux programmes de $\mathcal{T}$, il existe $p \circ q$ tel que

$$[\![p \circ q]\!]_{\mathcal{T}} = [\![p]\!]_{\mathcal{T}} \circ [\![q]\!]_{\mathcal{T}}$$

Il existe aussi un programme $\mathsf{compo}_{\mathcal{T}}$ tel que $[\![\mathsf{compo}_{\mathcal{T}}]\!]_{\mathcal{T}}(\overline{p}.\overline{q}.\mathbf{d}) = (\overline{p \circ q}.\mathbf{d})$.

D'autre part, on sait qu'il existe une fonction de spécialisation $\mathbf{spec}^{\mathcal{T}}$ et un spécialiseur $\mathsf{spec}_{\mathcal{T}}^{\mathcal{T}}$ dans les machines de Turing.

Comme $\mathcal{L}$ est $\mathcal{T}$-complet, il existe $\mathsf{int}_{\mathcal{L}}^{\mathcal{T}}$ un interpréteur de $\mathcal{T}$ dans $\mathcal{L}$. En effet, ce programme existe par l'existence d'un programme dans $\mathcal{L}$ calculant la même chose que $\mathsf{univ}_{\mathcal{T}}$.

Soient p et q deux programmes de $\mathcal{L}$. Notons $p' \stackrel{\text{def}}{=} \mathbf{spec}^{\mathcal{T}}(\mathsf{int}^{\mathcal{L}}_{\mathcal{T}}, \overline{p})$, de même pour q'. On a alors évidemment $[\![p']\!]_{\mathcal{T}} = [\![p]\!]_{\mathcal{L}}$ et $[\![q']\!]_{\mathcal{T}} = [\![q]\!]_{\mathcal{L}}$.

Posons $\mathsf{comp}^{\mathcal{L}\to\mathcal{T}}_{\mathcal{T}} \stackrel{\text{def}}{=} \mathbf{spec}^{\mathcal{T}}(\mathsf{spec}^{\mathcal{T}}_{\mathcal{T}}, \mathsf{int}^{\mathcal{L}}_{\mathcal{T}})$ et $\mathsf{comp}^{\mathcal{T}\to\mathcal{L}}_{\mathcal{T}} = \mathbf{spec}^{\mathcal{T}}(\mathsf{spec}^{\mathcal{L}}_{\mathcal{T}}, \mathsf{int}^{\mathcal{T}}_{\mathcal{L}})$. On a donc

$$\begin{array}{rcl} \left[\!\left[\mathsf{comp}^{\mathcal{L}\to\mathcal{T}}_{\mathcal{T}}\right]\!\right]_{\mathcal{T}}(\overline{p}.D) & = & \overline{p'}.D \\ \left[\!\left[\mathsf{comp}^{\mathcal{L}\to\mathcal{T}}_{\mathcal{T}}\right]\!\right]_{\mathcal{T}}(\overline{q}.D) & = & \overline{q'}.D \end{array}$$

Posons $\mathsf{comp}^{\mathcal{L}\to\mathcal{T}^2}_{\mathcal{T}} \stackrel{\text{def}}{=} \mathsf{switch}_{\mathcal{T}} \circ \mathsf{comp}^{\mathcal{L}\to\mathcal{T}}_{\mathcal{T}} \circ \mathsf{switch}_{\mathcal{T}} \circ \mathsf{comp}^{\mathcal{L}\to\mathcal{T}}_{\mathcal{T}}$. On a alors

$$\left[\!\left[\mathsf{comp}^{\mathcal{L}\to\mathcal{T}^2}_{\mathcal{T}}\right]\!\right](\overline{p}.\overline{q}.\mathbf{d}) = (p'.q'.\mathbf{d})$$

Et enfin, posons $\mathsf{compo}_{\mathcal{L}} \stackrel{\text{def}}{=} \mathbf{spec}^{\mathcal{L}}(\mathsf{int}^{\mathcal{T}}_{\mathcal{L}}, \overline{\mathsf{comp}^{\mathcal{T}\to\mathcal{L}}_{\mathcal{T}} \circ \mathsf{compo}_{\mathcal{T}} \circ \mathsf{comp}^{\mathcal{L}\to\mathcal{T}^2}_{\mathcal{T}}})$. On a donc

$$\begin{array}{rcl} [\![\mathsf{compo}_{\mathcal{L}}]\!]_{\mathcal{L}}(\overline{p}.\overline{q}.\mathbf{d}) & = & \left[\!\left[\mathsf{comp}^{\mathcal{T}\to\mathcal{L}}_{\mathcal{T}} \circ \mathsf{compo}_{\mathcal{T}} \circ \mathsf{comp}^{\mathcal{L}\to\mathcal{T}^2}_{\mathcal{T}}\right]\!\right]_{\mathcal{T}}(\overline{p}.\overline{q}.\mathbf{d}) \\ & = & \left[\!\left[\mathsf{comp}^{\mathcal{T}\to\mathcal{L}}_{\mathcal{T}} \circ \mathsf{compo}_{\mathcal{T}}\right]\!\right]_{\mathcal{T}}(\overline{p'}.\overline{q'}.\mathbf{d}) \\ & = & \left[\!\left[\mathsf{comp}^{\mathcal{T}\to\mathcal{L}}_{\mathcal{T}}\right]\!\right]_{\mathcal{T}}(\overline{p' \circ q'}.\mathbf{d}) \\ & = & \overline{\mathbf{spec}^{\mathcal{L}}(\mathsf{int}^{\mathcal{T}}_{\mathcal{L}}, \overline{p' \circ q'})}.\mathbf{d} \end{array}$$

Or

$$\begin{array}{rcl} \left[\!\left[\mathbf{spec}^{\mathcal{L}}(\mathsf{int}^{\mathcal{T}}_{\mathcal{L}}, \overline{p' \circ q'})\right]\!\right]_{\mathcal{L}} & = & [\![p' \circ q']\!]_{\mathcal{T}} \\ & = & [\![p']\!]_{\mathcal{T}} \circ [\![q']\!]_{\mathcal{T}} \\ & = & [\![p]\!]_{\mathcal{L}} \circ [\![q]\!]_{\mathcal{L}} \end{array}$$

Le programme $\mathbf{spec}^{\mathcal{L}}(\mathsf{int}^{\mathcal{T}}_{\mathcal{L}}, \overline{p' \circ q'}) \in \mathcal{P}$ est donc un programme composé de p et q et $\mathsf{compo}_{\mathcal{L}}$ est un programme calculant la composition dans $\mathcal{L}$. □

1.3.6 Compilation

Soient deux langages $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{D}, [\![]\!]_1)$ et $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{D}, [\![]\!]_2)$. Une fonction $\mathbf{comp}^{\mathcal{L}_2\to\mathcal{L}_1} : \mathcal{P}_2 \to \mathcal{P}_1$ (notée $\mathbf{comp}^{2\to1}$ dans la suite) est une *fonction de compilation* si

$$\forall d \in \mathcal{D}, \forall p_2 \in \mathcal{P}_2, \forall \mathbf{d} \in \mathcal{D}^* \quad \left[\!\left[\mathbf{comp}^{2\to1}(p_2)\right]\!\right]_1(d.\mathbf{d}) = [\![p_2]\!]_2(d.\mathbf{d})$$

On dira que $\mathbf{comp}^{2\to1}$ est implémentée dans le langage $\mathcal{L}_0 = (\mathcal{P}_0, \mathcal{D}, [\![]\!]_0)$ par le programme $\mathsf{comp}^{\mathcal{L}_2\to\mathcal{L}_1}_{\mathcal{L}_0} \in \mathcal{P}_0$ (noté $\mathsf{comp}^{2\to1}_0$ dans la suite) si

$$\left[\!\left[\mathsf{comp}^{2\to1}_0\right]\!\right]_0(\overline{p_2}.\mathbf{d}) = \overline{\mathbf{comp}^{2\to1}(p_2)}.\mathbf{d}$$

1.4 Théorie de la programmation

Dans cette section nous allons présenter les éléments de calculabilité dont nous allons avoir besoin dans le reste de cette thèse.

1.4.1 Projections de Futamura

Les notions de spécialisation et d'interprétation peuvent être mises en relation avec celle de compilation ([Fut83 ; JGS93 ; Jon97]). Soient quatre langages $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$ et $\mathcal{L}_4$ tous acceptables ($\mathcal{L}_i = (\mathcal{P}_i, \mathcal{D}, [\![]\!]_i)$, pour $i \in [\![1,4]\!]$). Pour des raisons de lisibilité, on a fixé un domaine de calcul identique $\mathcal{D}$ pour tous les langages. Par la Propriété 1.3.6 qu'il existe un interpréteur et un spécialiseur $\mathsf{int}_i^j, \mathsf{spec}_i^j \in \mathcal{P}_i$de $\mathcal{L}_j$ dans $\mathcal{L}_i$.

On va montrer qu'il est possible de construire une fonction de compilation et un compilateur à partir des interpréteurs et des spécialiseurs.

1.4.1.1 Fonction de compilation

On va construire une fonction de compilation à partir d'un interpréteur et d'une fonction de spécialisation. On dispose d'une fonction de spécialisation $\mathbf{spec}^3$. Soit un programme $p_4 \in \mathcal{P}_4$. Alors $p_3 \stackrel{\text{def}}{=} \mathbf{spec}^3(\mathsf{int}_3^4, \overline{p_4}) \in \mathcal{P}_3$ a la même sémantique que p_4, c'est à dire

$$\forall d \in \mathcal{D}, \mathbf{d} \in \mathcal{D}^* \quad [\![p_3]\!]_3 (d.\mathbf{d}) = [\![p_4]\!]_4 (d.\mathbf{d}).$$

En effet,

$$[\![\mathbf{spec}^3(\mathsf{int}_3^4, \overline{p_4})]\!]_3 (d.\mathbf{d}) = [\![\mathsf{int}_3^4]\!]_3 (\overline{p_4}.d.\mathbf{d}) = [\![p_4]\!]_4 (d.\mathbf{d})$$

On a donc défini une fonction de compilation

$$\mathbf{comp}^{4\to3} \stackrel{\text{def}}{=} p_4 \mapsto \mathbf{spec}^3(\mathsf{int}_3^4, \overline{p_4}).$$

1.4.1.2 Compilateur et fonction de génération de compilateurs

On va construire un compilateur et une fonction de génération de compilateurs paramétrés par un interpréteur. Posons $\mathsf{comp}_2^{4\to3} \stackrel{\text{def}}{=} \mathbf{spec}^2(\mathsf{spec}_2^3, \overline{\mathsf{int}_3^4})$. Le programme $\mathsf{comp}_2^{4\to3}$ est bien une implémentation de $\mathbf{comp}^{4\to3}$ dans $\mathcal{L}_2$ car, pour tout $p_4 \in \mathcal{P}_4$,

$$\begin{aligned}
[\![\mathsf{comp}_2^{4\to3}]\!]_2 (\overline{p_4}.\mathbf{d}) &= \left[\!\!\left[\mathbf{spec}^2(\mathsf{spec}_2^3, \overline{\mathsf{int}_3^4})\right]\!\!\right]_2 (\overline{p_4}.\mathbf{d}) \\
&= [\![\mathsf{spec}_2^3]\!]_2 (\overline{\mathsf{int}_3^4}.\overline{p_4}.\mathbf{d}) \\
&= \overline{\mathbf{spec}^3(\mathsf{int}_3^4, \overline{p_4})}.\mathbf{d} \\
&= \overline{\mathbf{comp}^{4\to3}(p_4)}.\mathbf{d}
\end{aligned}$$

On a donc défini une fonction de génération de compilateurs du langage $\mathcal{L}_4$ dans $\mathcal{L}_3$ écrit dans le langage $\mathcal{L}_2$ à partir d'un interpréteur de $\mathcal{L}_4$ écrit en $\mathcal{L}_3$

$$\mathbf{gen}^{3\overset{2}{\to}4} \stackrel{\text{def}}{=} \mathsf{int}_3^4 \mapsto \mathbf{spec}^2(\mathsf{spec}_2^3, \overline{\mathsf{int}_3^4})$$

1.4.1.3 Génération de compilateurs

La fonction de génération de compilateur est calculable. En continuant le principe d'enchaînement des spécialiseurs, on peut écrire dans le langage $\mathcal{L}_1$ une implémentation $\mathsf{gen}_1^{3\overset{2}{\to}4}$ de $\mathbf{gen}^{3\overset{2}{\to}4}$ telle que

$$\left[\!\!\left[\mathsf{gen}_1^{3\overset{2}{\to}4}\right]\!\!\right]_1 (\overline{\mathsf{int}_3^4}.\mathbf{d}) = \overline{\mathbf{gen}^{3\overset{2}{\to}4}(\mathsf{int}_3^4)}.\mathbf{d}$$

On pose

$$\mathsf{gen}_1^{3\xrightarrow{2}4} \stackrel{\text{def}}{=} \mathbf{spec}^1(\mathsf{spec}_1^2, \overline{\mathsf{spec}_2^3})$$

En effet,

$$\begin{aligned}
\left[\!\left[\mathsf{gen}_1^{3\xrightarrow{2}4}\right]\!\right]_1 (\overline{\mathsf{int}_3^4}.\mathbf{d}) &= \left[\!\left[\mathbf{spec}^1(\mathsf{spec}_1^2, \overline{\mathsf{spec}_2^3})\right]\!\right]_1 (\overline{\mathsf{int}_3^4}.\mathbf{d}) \\
&= \underline{[\![\mathsf{spec}_1^2]\!]_1 (\overline{\mathsf{spec}_2^3}.\overline{\mathsf{int}_3^4}.\mathbf{d})} \\
&= \underline{\mathbf{spec}^2(\mathsf{spec}_2^3, \overline{\mathsf{int}_3^4}).\mathbf{d}} \\
&= \mathbf{gen}^{3\xrightarrow{2}4}(\mathsf{int}_3^4).\mathbf{d}
\end{aligned}$$

Le programme $\mathsf{gen}_1^{3\xrightarrow{2}4}$ est donc une implémentation de $\mathbf{gen}^{3\xrightarrow{2}4}$ dans le langage $\mathcal{L}_1$.

1.4.2 Théorème de récursion de Kleene

Pour simplifier les notations, dans cette partie on ne travaillera que dans un seul langage, mais les résultats sont exprimables dans un cadre plus général où p et e sont dans des langages différents.

Théorème 1.4.1 (Kleene). *Soit $\mathcal{L} = (\mathcal{P}, \mathcal{D}, [\![]\!])$ un langage acceptable. Soit $p \in \mathcal{P}$ un programme. Il existe un programme $e \in \mathcal{P}$ satisfaisant l'équation de* Kleene *associée à p*

$$\forall \mathbf{d} \in \mathcal{D}^* \quad [\![e]\!](\mathbf{d}) = [\![p]\!](\overline{e}.\mathbf{d})$$

Intuitivement, ce théorème affirme que l'on peut construire un programme (réflexif) e à partir d'une spécification p *a priori* non réflexive.

Démonstration. On s'inspirera de la preuve de Jones dans [Jon97]. On dispose d'un programme dupl dupliquant son entrée (Propriété 1.3.2),

$$[\![\mathsf{dupl}]\!](d.\mathbf{d}) = d.d.\mathbf{d}.$$

Posons $\mathsf{omega} \stackrel{\text{def}}{=} \mathsf{spec} \circ \mathsf{dupl}$. On a donc

$$[\![\mathsf{omega}]\!](\overline{p}.\mathbf{d}) = [\![\mathsf{spec}]\!](\overline{p}.\overline{p}.\mathbf{d}) = \overline{\mathbf{spec}(p, \overline{p})}.\mathbf{d}$$

Enfin, posons

$$e \stackrel{\text{def}}{=} \mathbf{spec}(p \circ \mathsf{omega}, \overline{p \circ \mathsf{omega}}).$$

Alors

$$\begin{aligned}
[\![e]\!](\mathbf{d}) &= [\![\mathbf{spec}(p \circ \mathsf{omega}, \overline{p \circ \mathsf{omega}})]\!](\mathbf{d}) \\
&= [\![p \circ \mathsf{omega}]\!](\overline{p \circ \mathsf{omega}}.\mathbf{d}) \\
&= [\![p]\!]([\![\mathsf{omega}]\!](\overline{p \circ \mathsf{omega}}.\mathbf{d})) \\
&= [\![p]\!](\overline{\mathbf{spec}(p \circ \mathsf{omega}, \overline{p \circ \mathsf{omega}})}.\mathbf{d}) \\
&= [\![p]\!](\overline{e}.\mathbf{d})
\end{aligned}$$

□

On notera que la solution e est *calculable*[2], c'est à dire qu'il existe un programme calculant e à partir de $\overline{p}$.

2. grâce notamment à l'hypothèse 1.3.7 qui dit que la composition est calculable

Théorème 1.4.2. *Il existe une infinité de programmes e satisfaisant l'équation de* Kleene *associée à p.*

La preuve de ce théorème est inspirée de celles de Rogers [Rog87] ou de Odifreddi [Odi89] pour l'existence d'une fonction de *padding*.

Démonstration. Soit $p \in \mathcal{P}$. Posons $\mathbb{K}_p \subseteq \mathcal{P}$ l'ensemble des solutions e de l'équation de Kleene

$$\forall \mathbf{d} \in \mathcal{D}^*, \quad [\![p]\!]\,(\overline{e}.\mathbf{d}) = [\![e]\!]\,(\mathbf{d})$$

Supposons que cet ensemble soit fini. Il existe donc une fonction partielle semi-calculable $f : \mathcal{D}^* \to_\perp \mathcal{D}^*$ telle que $\forall e \in \mathbb{K}_p, f \neq [\![e]\!]$. De plus on peut construire un programme $q \in \mathcal{P}$ tel que

$$\begin{array}{rcll} [\![q]\!]\,(\overline{x}.\mathbf{d}) & = & [\![p]\!]\,(\overline{x}.\mathbf{d}) & \text{si } x \notin \mathbb{K}_p \\ & = & f(\mathbf{d}) & \text{sinon} \end{array}$$

Posons h une solution de l'équation de Kleene associée à q

$$\forall \mathbf{d} \in \mathcal{D}^*, \quad [\![q]\!]\,(\overline{h}.\mathbf{d}) = [\![h]\!]\,(\mathbf{d})$$

— Si $h \notin \mathbb{K}_p$, alors $[\![q]\!]\,(\overline{h}.\mathbf{d}) = [\![p]\!]\,(\overline{h}.\mathbf{d}) = [\![h]\!]\,(\mathbf{d})$. Donc $h \in \mathbb{K}_p$.

— Si $h \in \mathbb{K}_p$, alors $[\![q]\!]\,(\overline{h}.\mathbf{d}) = f(\mathbf{d}) = [\![h]\!]\,(\mathbf{d})$. Donc $f = [\![h]\!]$.

Dans les deux cas, on tombe sur une contradiction. Donc l'ensemble $\mathbb{K}_p$ est infini. □

1.4.3 Utilisation pratique du Second Théorème de Récursion

Le second théorème de récursion de Kleene (Section 1.4.2) peut être interprété en terme de comportements réflexifs. L'article [Mos10] présente différentes applications du théorème de récursion. Un premier exemple d'utilisation est la preuve d'existence de *quines* dans tout langage acceptable, c'est à dire de programmes renvoyant leur propre code. Si on prend $p = \mathsf{id}$, alors $[\![p]\!]\,(\overline{e}.\mathbf{d}) = \overline{e}.\mathbf{d}$, et le second théorème de récursion fournit un programme e tel que

$$[\![e]\!]\,(\mathbf{d}) = \overline{e}.\mathbf{d}$$

Dans [Mar12], le théorème est utilisé comme compilateur de programmes auto-répliquants. Par exemple, on cherche à construire un programme v respectant la spécification $\mathfrak{S}$, c'est à dire un programme construisant le programme $\mathcal{B}(p, v)$ à partir du code du programme initial p et du code de v. Par exemple, si v est un virus informatique, $\mathcal{B}(p, v)$ est le code du programme p infecté par v.

La spécification $\mathfrak{S}$ construit le code $\mathcal{B}(p, v)$ à partir de p et v :

$$[\![\mathfrak{S}]\!]\,(\overline{v}, \overline{p}) = \mathcal{B}(p, v).$$

D'autre part, le programme v répond à cette spécification, c'est à dire que

$$[\![v]\!]\,(\overline{p}) = [\![\mathfrak{S}]\!]\,(\overline{v}, \overline{p}).$$

Le programme v peut donc être construit à partir du Second Théorème de Récursion. Dans le cas de virus informatique, on a donc construit un programme qui infecte le code d'un autre programme à partir de son propre code.

1.5 Conclusion

Nous avons défini une première partie des notions omniprésentes dans cette thèse, notamment ce qu'était un langage de programmation et quel était le sens des objets utilisés. On a présenté deux programmes essentiels dans un langage de programmation : l'interpréteur et le spécialiseur.

D'autre part, notre définition d'un langage de programmation met en exergue la distinction programmes/données. Cette différence est fondamentale dans la compréhension de la réflexion, comme on le verra dans la suite. Mais ce chapitre a déjà commencé à introduire des phénomènes de conversion entre programmes et données, grâce aux interpréteurs notamment.

Un autre moyen a été présenté, il s'agit du Théorème de Récursion de Kleene. En effet, ce théorème démontre l'existence de programmes dont l'exécution dépend de leur propre code, c'est à dire, voyant leur code comme une donnée.

Chapitre 2

Réflexion

Les deux premières parties de ce chapitre sont des parties introductives, mettant en évidence les éléments en jeu dans notre sujet. La troisième partie donne un début de formalisation des concepts abordés dans les parties précédentes.

2.1 Auto-référence

Dans cette partie on va s'intéresser plus particulièrement à l'*auto-référencement*, c'est à dire la capacité d'un programme de s'observer lui-même (ce qu'on appellera dans la suite *réification*). La présentation est inspirée par Smullyan [Smu84].

On souhaite comprendre l'expressivité d'un langage possédant des capacités réflexives. Dans un premier temps, on va simplement s'intéresser aux manières qu'a un langage de faire référence à lui-même. Dans tout ce qui suit, les notions seront illustrées dans la langue française.

2.1.1 Dualité phrase/lecteur

Deux entités sont d'emblée mises en avant. D'une part, la *phrase*, ou encore l'*instruction*, suivant une syntaxe donnée. Une phrase peut être par exemple

$$\textit{Écrire } \langle \textit{Bonjour} \rangle \tag{2.1}$$

La deuxième entité est ce qui donne du sens à la première. En français, il s'agit du *lecteur*. On parlera aussi d'*interpréteur* dans le cas général d'un langage de programmation. Cette entité est responsable de l'exécution de la phrase.

En français, le lecteur lisant la phrase (2.1) va écrire "*Bonjour*". Il est à noter ici l'usage de la syntaxe ⟨ et ⟩ servant à délimiter une partie non interprétée (appelons ça les *citations*) par le lecteur de la phrase. Lorsque qu'une syntaxe est comprise entre deux chevrons, elle désigne un objet figé, syntaxique. Dans le cas contraire, la syntaxe désigne un objet exécutable, fonctionnel.

Dans la suite, on utilisera ⟨ et ⟩ comme éléments de syntaxe, et “ et ” comme méta-syntaxe délimitant les phrases du langage.

Ici, la sémantique de la phrase ne pose pas de problème, à partir du moment où l'on sait ce que veulent dire “*Écrire*” et “⟨*Bonjour*⟩”. Elle est *compositionnelle*.

On note qu'étant donné qu'une syntaxe peut apparaître soit comme une instruction à exécuter (comme “*Écrire*”) ou soit comme un objet syntaxique (comme “*Bonjour*”), selon qu'elle est entourée ou non par les chevrons ⟨ et ⟩, deux sémantiques sont possibles : (i) une sémantique extensionnelle (notée $[\![]\!]^e$) donnant la signification d'une syntaxe vue comme objet fonctionnel, c'est à dire comme une fonction dont on ne s'intéresse qu'au rapport entre ses entrées et ses sorties, (ii) un sémantique intentionnelle (notée $[\![]\!]^i$) donnant la signification d'une syntaxe vue comme un objet syntaxique, c'est à dire comme une structure de données. Par exemple, la sémantique intentionnelle d'une phrase peut être son numéro de Gödel. La sémantique extensionnelle de “⟨*Bonjour*⟩” est donnée par $[\![\langle Bonjour\rangle]\!]^e = [\![Bonjour]\!]^i$.

2.1.2 Exécution

Dans l'exemple précédent, on a vu que le lecteur se chargeait d'exécuter ce que la phrase lui indiquait. Parmi ces indications, on peut imaginer une phrase demandant d'exécuter une autre phrase. Par exemple la phrase

$$\textit{Faire } \langle \textit{Écrire } \langle \textit{Bonjour}\rangle\rangle \tag{2.2}$$

a la même sémantique que le premier exemple. Cependant, l'exécution se fait en deux fois, l'exécution de la phrase initiale, puis celle de la citation. Ici, la sémantique est toujours compositionnelle, mais elle demande de passer de l'interprétation syntaxique $[\![]\!]^i$ de “*Écrire* ⟨*Bonjour*⟩” à son interprétation fonctionnelle $[\![]\!]^e$, car l'instruction “*Faire*” demande d'exécuter la citation “⟨*Écrire* ⟨*Bonjour*⟩⟩”.

2.1.3 Quines/auto-référence

À présent on souhaite introduire un moyen pour un programme d'inspecter son propre code. Pour cela, on va chercher les programmes capables d'*écrire leur propre code*. De tels programmes s'appellent des *quines*. Par exemple, la phrase suivante est un quine en français

$$\textit{Écrire puis écrire entre chevrons } \langle \textit{Écrire puis écrire entre chevrons}\rangle \tag{2.3}$$

L'intérêt de tels programmes pour les phénomènes réflexifs est évident : pouvant lire tout ce qu'il écrit, il peut ainsi accéder à sa syntaxe et l'analyser selon ses besoins.

2.1.4 Sémantique contextuelle

Considérons maintenant la phrase

$$\textit{Écrire cette phrase} \tag{2.4}$$

L'exécution de cette phrase va écrire "*Écrire cette phrase*". On a introduit un nouvel élément de syntaxe : "*cette phrase*". Il permet la réflexivité en faisant référence à la phrase lue par le lecteur. C'est un moyen simple de créer des quines, comme dans la Section 2.1.3. Une phrase utilisant la syntaxe "*cette phrase*" en dehors d'une citation sera dite *indexicale*. Dans le cas contraire, elle sera dite *normale*.

Contrairement à l'exemple précédent, la sémantique de la phrase n'est pas définie à partir des sémantiques de ses éléments (elle n'est pas compositionnelle) car la sémantique de l'élément "*cette phrase*" n'est même pas définie en dehors de tout contexte. Au contraire, plongée dans un contexte, cette syntaxe le capture et le met à disposition du lecteur (interpréteur). On appellera *réification* ce phénomène. On va voir dans la section suivante comment donner un sens à cette syntaxe réflexive.

Il est à noter que si "*cette phrase*" apparaît dans une citation, le sens de la phrase reste bien défini, car on utilise sa signification syntaxique. Ainsi, la variable "*cette phrase*" ne fait référence à la phrase complète que lorsqu'elle apparaît hors d'une citation. Dit autrement, la portée d'une référence "*cette phrase*" ne dépasse pas les chevrons. Par exemple, les deux références de la phrase suivante ne représentent pas la même chose

$$\textit{Écrire cette phrase et faire } \langle \textit{Écrire cette phrase} \rangle. \tag{2.5}$$

La première fait référence à "*Écrire cette phrase et faire* $\langle$*Écrire cette phrase*$\rangle$" et la deuxième à "*Écrire cette phrase*". Si σ est une abréviation pour "*cette phrase*", la phrase "*Écrire* σ" peut être vue comme une notation implicite de "$\mathsf{phrase}\,\sigma.$*Écrire* σ", où $\mathsf{phrase}\,\sigma$ est un *lieur*. La phrase (2.5) est donc une version implicite de

$$\mathsf{phrase}\,\sigma.\ \textit{Écrire } \sigma \textit{ et faire } \langle \mathsf{phrase}\,\sigma'.\ \textit{Écrire } \sigma' \rangle. \tag{2.6}$$

2.1.5 Diagonalisation

Dans un premier temps, on transforme une phrase indexicale en phrase normale. Par exemple, reprenons la phrase (2.4). Cette phrase est équivalente à la phrase (qui n'est plus un quine)

$$\textit{Écrire } \langle \textit{Écrire cette phrase} \rangle \tag{2.7}$$

obtenue en replaçant "*cette phrase*" par "$\langle$*Écrire cette phrase*$\rangle$". Cette phrase est normale, car la référence "*cette phrase*" n'apparaît plus en dehors d'une citation. La sémantique de la phrase obtenue par diagonalisation est bien définie (compositionnelle) car les éléments "*Écrire*" et "$\langle$*Écrire cette phrase*$\rangle$" sont définissables dans n'importe quel contexte. Si on reprend le formalisme avec $\mathsf{phrase}\,\sigma$, le lieur phrase apparaît comme un lieur de point fixe : si p est une phrase, $\mathsf{phrase}\,\sigma.p$ est équivalente à $p[\sigma/\langle\,\mathsf{phrase}\,\sigma.p\rangle]$. On abordera cette idée dans la Section 7.4.5.1.

2.2 Concepts

Dans cette partie on se propose d'établir une liste des concepts introduits par la réflexion dans un langage de programmation.

2.2.1 But de la réflexion : passage de la sémantique extensionnelle à la sémantique intentionnelle

Lors de l'analyse de programmes, deux points peuvent intéresser. La première question est "Que fait ce programme ?", "Étant donnée une entrée, que va t'il renvoyer ?". L'intérêt porte ici uniquement sur les entrés et les sorties du programme, on le voit d'un point de vue extérieur. Le point de vue est *extensionnel*. La définition de la sémantique d'un langage de programmation donnée dans la partie précédente est un exemple de sémantique extensionnelle, car elle associe à tout programme une fonction partielle des données vers les données. C'est à dire que l'on ne s'intéresse qu'à l'action qu'a le programme sur les données en fonction de son entrée.

Dans la majorité des langages de programmation, un programme n'a accès qu'à la sémantique extensionnelle des autres programmes : il peut demander d'exécuter d'autres programmes sur des paramètres qu'il choisit, et récupérer le résultat produit par ces programmes. En revanche, il ne sait pas comment s'est déroulé le calcul. Par exemple, deux programmes différents calculant la même fonction lui seront indistinguables.

La deuxième façon d'étudier un programme est de répondre à la question "Comment fonctionne ce programme ?". La question n'est plus de savoir "*Que* fait un programme ?" mais "*Comment* le fait il ?". On parle de sémantique *intentionnelle*. Par exemple, des considérations de complexité, de taille du programme sont des arguments intentionnels.

L'intérêt de la réflexion est de donner au langage de programmation la possibilité d'accéder à la sémantique intentionnelle des programmes. Une façon simple est de donner accès à la syntaxe des programmes, de les faire apparaître comme une donnée habituelle.

2.2.2 Vocabulaire de la réflexion

Les notions abordées par les problèmes de réflexion sont décrites en partie par Smith dans [Smi83], ou dans [WF ; DMM88] qui sont des réécritures du travail de Smith. On donne ici une explication des termes que l'on va utiliser dans la suite.

2.2.2.1 Réflexion structurelle et procédurale

La *réflexion structurelle* permet de traiter les programmes comme des données (et *vice versa*), pour en changer la structure ou, au contraire, vérifier qu'elle n'a pas été corrompue.

La *réflexion procédurale* est la capacité accéder au processus de calcul permettant d'exécuter le programme. Par exemple, elle peut donner accès au contexte d'exécution du programme, qui peut être lu et modifié pour influencer le déroulement du calcul.

2.2.2.2 Réflexion et réification

Un langage réflexif met en jeu deux comportements. Le premier consiste à transformer un programme en donnée lisible et éditable. Ce phénomène s'appelle la *réification*. On passe d'un programme, c'est à dire un objet extensionnel (un programme est connu à partir du moment où on connait sa sémantique $[\![p]\!]$) à une donnée, un objet syntaxique, dont la sémantique est intentionnelle (syntaxique) (on voit une donnée comme une structure). Par exemple la numérotation de Gödel est une façon de réifier un programme.

Le deuxième phénomène est le processus inverse, c'est à dire qu'il consiste en la transformation d'une donnée en un programme. C'est la *réflexion*.

2.2.2.3 Tour de réflexions

On peut envisager un programme lançant l'exécution d'une donnée (réflexion), qui elle même lancera l'exécution d'une autre donnée, etc. Cette idée est la base de la *tour de réflexions* de Smith. Un programme p_n demande l'exécution d'une donnée d_n représentant le programme p_{n+1} s'exécutant sur la donnée d_{n+1}.

2.3 Interpréteur récursif

2.3.1 Définition

Définition 2.3.1 (Interpréteur récursif)**.** Soient deux langages acceptables $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{D}, [\![]\!]_1)$ et $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{D}, [\![]\!]_2)$. Un programme $\mathsf{int}\omega_1^2 \in \mathcal{P}_1$ est un interpréteur récursif de $\mathcal{L}_2$ dans $\mathcal{L}_1$ si

$$\forall p_2 \in \mathcal{P}_2, \mathbf{d} \in \mathcal{D}^* \quad [\![\mathsf{int}\omega_1^2]\!]_1 (\overline{p_2}.\mathbf{d}) = [\![p_2]\!]_2 (\overline{\mathsf{int}\omega_1^2}.\mathbf{d})$$

L'existence de tels programmes est fournie par le théorème de récursion. En effet, posons $\mathsf{switch} \in \mathcal{P}_1$ un programme tel que

$$[\![\mathsf{switch}]\!]_1 (a.b.\mathbf{d}) = (b.a.\mathbf{d})$$

Et posons e le point fixe de $\mathsf{int}_1^2 \circ \mathsf{switch}$ obtenu par le Théorème de Récursion. On a alors

$$\begin{array}{rcl} [\![e]\!]_1 (\overline{p_2}.\mathbf{d}) & = & [\![\mathsf{int}_1^2 \circ \mathsf{switch}]\!]_1 (\overline{e}.\overline{p_2}.\mathbf{d}) \\ & = & [\![\mathsf{int}_1^2]\!]_1 (\overline{p_2}.\overline{e}.\mathbf{d}) \\ & = & [\![p_2]\!]_2 (\overline{e}.\mathbf{d}) \end{array}$$

2.3.2 Exemple

Un interpréteur récursif réifie son propre code et le passe à l'exécution d'une donnée. Il en existe une infinité (Théorème 1.4.2). La donnée exécutée peut donc en extraire une information non triviale. Par exemple, elle peut en extraire un nouveau code à exécuter. Nous allons présenter cette application dans la suite. Ce n'est qu'un exemple présentant le genre d'information qu'un interpréteur récursif peut contenir.

2.3.2.1 Scénario

Soient deux langages $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{D}, [\![]\!]_1)$ et $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{D}, [\![]\!]_2)$. On va s'intéresser au passage du langage $\mathcal{L}_1$ au langage $\mathcal{L}_2$ grâce à un interpréteur récursif. On a $p_1 = \mathsf{int}\omega_1^2$ et on veut exécuter la donnée $\overline{p_2}$, où p_2 est un programme de $\mathcal{L}_2$.

Le programme p_2, une fois lancé, a accès au code de p_1. On utilise cette capacité pour passer à p_2 le programme p_1' de $\mathcal{L}_1$ qu'il devra exécuter après sa propre exécution.

On utilise pour cela l'infinité de programmes p_1 possibles. On encode "dans cette infinité" la continuation p'_1. On note $p_1 \stackrel{\text{def}}{=} \mathbf{push}_1^2(p'_1)$ l'interpréteur récursif encodant avec lui sa continuation p'_1.

Lorsque p_2 sera exécuté, il disposera du code $\overline{p_1} = \overline{\mathbf{push}_1^2(p')}$. Pour pouvoir lancer l'exécution de p'_1, il doit en premier lieu l'extraire de $\overline{p_1}$. Il utilise pour cela le programme $\mathsf{next}_2^1 \in \mathcal{P}_2$. Il obtient alors le code $\overline{p'_1}$ qui peut à loisir exécuter (grâce à un autre interpréteur, récursif ou non).

2.3.2.2 Interpréteur récursif paramétré

Soient deux langages acceptables $\mathcal{L}_1$ et $\mathcal{L}_2$.

Définition 2.3.2 (Interpréteur récursif avec continuation)**.** Une fonction injective $\mathbf{push}_1^2 : \mathcal{P}_1 \to \mathcal{P}_1$ est une *fonction de génération d'interpréteurs récursifs avec continuation* lorsque

$$\forall p'_1 \in \mathcal{P}_1, p_2 \in \mathcal{P}_2, \mathbf{d} \in \mathcal{D}^* \quad \left[\!\left[\mathbf{push}_1^2(p'_1)\right]\!\right]_1 (\overline{p_2}.\mathbf{d}) = [\![p_2]\!]_2 (\overline{\mathbf{push}_1^2(p'_1)}.\mathbf{d})$$

Pour tout p_1, $\mathbf{push}_1^2(p_1)$ est un interpréteur récursif. Le programme p_1 est la continuation de l'interpréteur récursif.

Définition 2.3.3 (Extracteur)**.** Un programme $\mathsf{next}_2^1 \in \mathcal{P}_2$ est un *extracteur* associé à la fonction $\mathbf{push}_1^2$ si

$$\left[\!\left[\mathsf{next}_2^1\right]\!\right]_2 (\overline{\mathbf{push}_1^2(p_1)}.\mathbf{d}) = \overline{p_1}.\mathbf{d}$$

La fonction de génération et l'extracteur permettent de tirer profit de la réification autorisée par les interpréteurs récursifs. La donnée $\overline{p_2}$ exécutée par l'interpréteur récursif peut extraire de l'information à partir du code de l'interpréteur : par exemple, elle peut exécuter un programme encodé dans l'interpréteur. Posons $p_2 = \mathsf{int}_2^1 \circ \mathsf{next}_2^1$; ce programme extrait la continuation p'_1 (grâce à next_2^1) avant de l'exécuter (grâce à int_2^1). Le programme $p_1 \stackrel{\text{def}}{=} \mathbf{spec}^1(\mathbf{push}_1^2(p'_1), \overline{\mathsf{int}_2^1 \circ \mathsf{next}_2^1})$ s'exécute de la manière suivante

$$\begin{aligned}
\left[\!\left[\mathbf{spec}^1(\mathbf{push}_1^2(p'_1), \overline{\mathsf{int}_2^1 \circ \mathsf{next}_2^1})\right]\!\right]_1 (\mathbf{d}) &= \left[\!\left[\mathbf{push}_1^2(p'_1)\right]\!\right]_1 (\overline{\mathsf{int}_2^1 \circ \mathsf{next}_2^1}.\mathbf{d}) \\
&= \left[\!\left[\mathsf{int}_2^1 \circ \mathsf{next}_2^1\right]\!\right]_2 (\overline{\mathbf{push}_1^2(p'_1)}.\mathbf{d}) \\
&= \left[\!\left[\mathsf{int}_2^1\right]\!\right]_2 (\overline{p'_1}.\mathbf{d}) \\
&= [\![p'_1]\!]_1 (\mathbf{d})
\end{aligned}$$

2.3.2.3 Preuves d'existence

On montre à présent que la fonction $\mathbf{push}_1^2$ et le programme next_2^1 existent.

Propriété 2.3.4. *Il existe une fonction de génération d'interpréteur récursif*

Démonstration. Ce résultat est une conséquence directe du Théorème 1.4.2 d'une part, et de l'infinité dénombrable de $\mathcal{P}$. □

Propriété 2.3.5. *Il existe une fonction* $\mathbf{push}_1^2$ *calculable*

Démonstration. On rappelle qu'un interpréteur récursif est solution de l'équation de Kleene associée à $\mathsf{int}_1^2 \circ \mathsf{switch}$. On note $\mathbb{K} \subseteq \mathcal{P}_1$ l'ensemble des solutions de cette équation.

L'ensemble $\mathcal{P}_1$ est effectivement dénombrable, c'est à dire qu'il existe une bijection totale calculable $\mathbb{N} \to \mathcal{P}_1$, ou, ce qui revient au même, qu'il existe une bijection totale calculable de $f : \mathcal{P}_1 \to \mathbb{N}$. Donc il suffit de montrer qu'il existe une fonction $g : \mathbb{N} \to \mathbb{K}$ calculable, car alors on pose $\mathbf{push}_1^2 = g \circ f$, que l'on sait être calculable par la Propriété 1.3.7.

Montrons qu'il existe une telle fonction g. Pour cela, on va montrer qu'il existe un programme trouvant un $n+1$-ème interpréteur récursif à partir d'un ensemble $\mathbb{K}^n \subset \mathbb{K}$ de n interpréteurs récursifs.

Notons dans une premier temps que la fonction semi-calculable **id** n'est calculée par aucun programme solution de cette équation. L'ensemble $\mathbb{K}^n$ nous permet de construire (de manière calculable) un programme $q_n \in \mathcal{P}_1$ tel que

$$\begin{array}{rcll} [\![q_n]\!]_1 (\overline{x}.\mathbf{d}) & = & [\![\mathsf{int}_1^2 \circ \mathsf{switch}]\!]_1 (\overline{x}.\mathbf{d}) & \text{si } x \notin \mathbb{K}^n \\ & = & \mathbf{id}(\mathbf{d}) & \text{sinon} \end{array}$$

Posons $h_{n+1} \in \mathcal{P}_1$ une solution de l'équation de Kleene associée à q_n.

$$\forall d \in \mathcal{D}, \quad [\![q_n]\!]_1 (\overline{h_{n+1}}.\mathbf{d}) = [\![h_{n+1}]\!]_1 (\mathbf{d})$$

Étant donné que la construction du point fixe de Kleene est calculable, h_{n+1} a été construit de manière calculable.

D'une part, $h_{n+1} \notin \mathbb{K}^n$ car sinon, pour toute liste de données $\mathbf{d}$, $[\![q_n]\!]_1 (\overline{h_{n+1}}.\mathbf{d}) = \mathbf{id}(\mathbf{d}) = [\![h_{n+1}]\!]_1 (\mathbf{d})$ et donc $\mathbf{id} = [\![h_{n+1}]\!]_1$, impossible car $h_{n+1} \in \mathbb{K}^n$. De plus, $h_{n+1} \in \mathbb{K}$ car $[\![q_n]\!]_1 (\overline{h_{n+1}}.\mathbf{d}) = [\![\mathsf{int}_1^2 \circ r]\!]_1 (\overline{h_{n+1}}.\mathbf{d}) = [\![h_{n+1}]\!]_1 (\mathbf{d})$.

On a donc construit de manière calculable h_{n+1} qui est un $n+1$-ème interpréteur récursif. □

Propriété 2.3.6. *Il existe un programme* $\mathsf{next}_2^1 \in \mathcal{P}_2$ *tel que*

$$\forall p_1 \in \mathcal{P}_1, [\![\mathsf{next}_2^1]\!]_2 (\overline{\mathbf{push}_1^2(p_1)}.\mathbf{d}) = \overline{p_1}.\mathbf{d}$$

Démonstration. La fonction $\mathbf{push}_1^2$ est calculable et injective, donc sa réciproque est une fonction semi-calculable et injective. □

2.4 Tour de réflexions

Smith introduit le concept de *tour de réflexions*, qui est un moyen de décrire la dynamique réflexive d'un programme. Ce travail va utiliser cette représentation lui aussi. Les articles [WF ; DMM88] donnent des exemples d'utilisation de cette représentation.

2.4.1 Définition

Définition 2.4.1 (Tour de réflexions)**.** Soient deux langages acceptables $\mathcal{L}_1$ et $\mathcal{L}_2$. Le programme $p_1 \in \mathcal{P}_1$ est une *tour de réflexions* de fils $p_2 \in \mathcal{P}_2$ lorsqu'il existe un interpréteur

récursif $\mathsf{int}\omega_1^2 \in \mathcal{P}_1$ et une fonction de spécialisation $\mathbf{spec}^1$ tels que

$$p_1 = \mathbf{spec}^1(\mathsf{int}\omega_1^2, \overline{p_2})$$

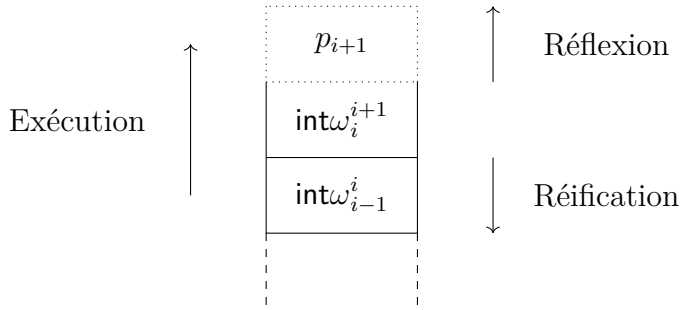
On notera $p_1 \rightsquigarrow p_2$ et se lit p_1 *réfléchit* p_2.

Propriété 2.4.2. *Si* $p_1 \rightsquigarrow p_2$, *alors* $[\![p_1]\!]_1(\mathbf{d}) = [\![p_2]\!]_2(\mathsf{int}\omega_1^2.\mathbf{d})$

On peut concevoir un programme en terme de tour de réflexions :

$$p_1 \rightsquigarrow p_2 \rightsquigarrow \cdots \rightsquigarrow p_n$$

Dans ce cas, $\forall i \leqslant n, [\![p_1]\!]_1(\mathbf{d}) = [\![p_{i+1}]\!]_{i+1}(\overline{\mathsf{int}\omega_i^{i+1}}.\overline{\mathsf{int}\omega_{i-1}^{i}}.\cdots.\overline{\mathsf{int}\omega_1^2}.\mathbf{d})$. Le programme p_{i+1} a donc accès à toute la pile des réifications des interpréteurs récursifs s'étant exécutés avant lui.



Les sections suivantes donnent différentes utilisations de cette vue en tour de réflexion.

2.4.2 Contrôle de l'exécution

Dans la Section 2.3.2 on a donné un exemple d'utilisation des interpréteurs récursifs comme moyen de passer à la donnée exécutée une version réifiée de la continuation du programme complet. Cet exemple passe dans les tours de réflexion : le programme p_{i+1} a accès aux versions réifiées de tous les interpréteurs récursifs des programmes p_1, ..., p_i. Il peut donc potentiellement en extraire toutes les continuations p'_j encodées dans ces interpréteurs $\mathsf{int}\omega_j^{j+1} = \mathbf{push}_j^{j+1}(p'_j)$. La liste $\mathsf{int}\omega_1^2, \cdots, \mathsf{int}\omega_i^{i+1}$ introduit une *meta-continuation* $p'_1, \cdots, p'_i$ à laquelle a accès p_{i+1}. Cette idée sera mise en pratique dans le Chapitre 11.

2.4.3 Mesure de la réflexion

La décomposition d'un programme réflexif en tour de réflexions permet de donner un sens au programme au niveau de ses comportements réflexifs, c'est à dire de mettre en valeur "comment" ce programme est réflexif.

2.4.3.1 Segmenter l'exécution en programmes non réflexifs

Entre p_i et p_{i+1} se trouve une spécialisation d'un interpréteur récursif. Dans la spécialisation se cache la *charge* c_i de p_i, c'est à dire le calcul effectué par p_i avant de passer la main à p_{i+1}. Pour faire apparaître cette charge, on peut écrire p_1 avec un spécialiseur $\mathbf{spec}^{1'}$ que l'on peut supposer "de charge vide"

$$p_1 = \mathbf{spec}^{1'}(\mathsf{int}\omega_1^2 \circ c_1, d_1)$$

avec $[\![c_1]\!]_1 (d_1.\mathbf{d}) = \overline{p_2}.\mathbf{d}$. L'idée est de faire porter à c_1 les *calculs non réflexifs*, puis d'appeler p_2 en utilisant l'interpréteur récursif. La décomposition de p_1 en une charge suivie d'une réflexion *segmente* l'exécution en calculs successifs de plusieurs charges, séparés par des réflexions.

2.4.3.2 Quantifier la réflexion

L'itération charge/réflexion donne au programme une vue *nivelée*, comme dans une tour. Cette interprétation "graphique" de l'exécution permet d'extraire de l'exécution uniquement les informations relevant de la réflexion. Une exécution peut être abstraite en ne gardant que cette information. On verra une mise en œuvre de cette idée dans le Chapitre 4.

2.4.3.3 Simplification de la tour

Dans le cas où on ne s'intéresse qu'aux réflexions, on peut remplacer l'interpréteur récursif par un interpréteur simple

$$p_1 = \mathbf{spec}^1(\mathsf{int}_1^2, p_2)$$

Dans ce cas, une fois exécuté, p_2 n'a pas accès au code de int_1^2. Il n'y a donc *pas de réification*.

2.4.4 Obfuscation de programmes

2.4.4.1 Définition de l'obfuscation

L'*obfuscation* de programme consiste à modifier le code d'un programme pour en cacher la sémantique sans toutefois la modifier. Plusieurs méthodes sont possibles (ajout de codes morts, modification du flot d'exécution, ...).

Définition 2.4.3 (Obfuscation)**.** On dit que la fonction $\mathbf{obf} : \mathcal{P} \to \mathcal{P}$ est une *fonction d'obfuscation* lorsque

$$\forall p, \quad \mathbf{obf}(p) \neq p \ \wedge \ [\![\mathbf{obf}(p)]\!] = [\![p]\!]$$

L'existence de telles fonctions (et de leur infinité) peut être prouvée par le théorème de récursion (voir[Rog87] ou [Odi89]). Plus simplement, dans une machine de Turing par exemple, on peut construire cette fonction en ajoutant des états non accessibles.

On peut espérer construire une fonction d'obfuscation à partir d'une machine universelle et d'une fonction de spécialisation : posons

$$\mathbf{obf} : p \mapsto \mathbf{spec}(\mathsf{univ}, \overline{p})$$

Alors on a bien $[\![\mathbf{obf}(p)]\!] = [\![p]\!]$. En pratique, $\mathbf{obf}(p)$ est différent de p et il est même difficile de retrouver p à partir de $\mathbf{obf}\, p$.

Giacobazzi, Jones et Mastroeni [GJM12] proposent d'utiliser cette décomposition en une fonction de spécialisation et d'une machine universelle pour construire des fonctions d'obfuscation. Ils utilisent pour cela des machines universelles "tordues" dans leur façon d'interpréter un programme. Ils montrent que cette méthode produit effectivement des programmes différents et difficiles à comprendre (la notion de "difficile à comprendre" est définie à partir d'interprétation abstraite)

Une autre façon de faire est d'utiliser une fonction de spécialisation "tordue". Supposons que l'on dispose d'une fonction de chiffrement $\mathbf{enc} : \mathcal{D} \to \mathcal{D}$ et d'un programme dec tel que $[\![\mathsf{dec}]\!]\,(\mathbf{enc}(d).\mathbf{d}) = d.\mathbf{d}$. Si l'on dispose d'une fonction de spécialisation $\mathbf{spec}$, on peut en construire une nouvelle $\mathbf{spec}\,' : (p, d) \mapsto \mathbf{spec}(p \circ \mathsf{dec}, \mathbf{enc}(d))$.

$$\begin{array}{rcl} [\![\mathbf{spec}\,'(p,d)]\!]\,(\mathbf{d}) & = & [\![\mathbf{spec}\,(p \circ \mathsf{dec}, \mathbf{enc}(d))]\!]\,(\mathbf{d}) \\ & = & [\![p \circ \mathsf{dec}]\!]\,(\mathbf{enc}(d).\mathbf{d}) \\ & = & [\![p]\!]\,(d.\mathbf{d}) \end{array}$$

On obtient alors une fonction d'obfuscation avec $p \mapsto \mathbf{spec}\,'(\mathsf{univ}, \overline{p})$. Cette méthode est, en pratique, souvent utilisée pour masquer le code d'un programme.

2.4.4.2 Obfuscation et réflexion

Au lieu d'utiliser une machine universelle, on peut se ramener à notre modèle de réflexion en utilisant un interpréteur récursif :

$$\mathbf{obf} : p \mapsto \mathbf{spec}(\mathsf{int}\omega, \overline{p \circ \mathsf{pop}})$$

La composée de ces fonctions d'obfuscations (dépendant de $\mathbf{spec}$ et de $\mathsf{int}\omega$) peut donc être vue comme une *tour de réflexions*.

$$\begin{array}{rcl} [\![\mathbf{spec}(\mathsf{int}\omega, \overline{p_2 \circ \mathsf{pop}})]\!]\,(\mathbf{d}) & = & [\![\mathsf{int}\omega]\!]\,(\overline{p_2 \circ \mathsf{pop}}.\mathbf{d}) \\ & = & [\![p_2 \circ \mathsf{pop}]\!]\,(\overline{\mathsf{int}\omega}.\mathbf{d}) \\ & = & [\![p_2]\!]\,(\mathbf{d}) \end{array}$$

Si on pose $p_1 \stackrel{\text{def}}{=} \mathbf{spec}(\mathsf{int}\omega, \overline{p_2 \circ \mathsf{pop}})$, selon les notations de la Section 2.4.1, on a

$$p_1 \rightsquigarrow p_2 \circ \mathsf{pop}$$

Le point de vue est donc ici de voir les couches d'obfuscations comme les différents niveaux d'une tour de réflexions.

2.5 Conclusion

La réflexion est la capacité d'un programme d'accéder à la sémantique intentionnelle d'autres programmes (c'est à dire, *comment* ils calculent). Deux phénomènes sont possibles : soit le programme *réfléchit* une donnée, c'est à dire la transforme en programme, soit il réifie un programme, c'est à dire le transforme en donnée.

On a donné une formalisation possible de la réflexion dans n'importe quel langage de programmation, en utilisant les notions élémentaires de la programmation (spécialiseur, interpréteur, Théorème de Récursion). Dans ce cadre, on a défini une *tour de réflexions* qui est l'itération de réflexions successives.

On a donné plusieurs applications de cette tour. Elle permet un contrôle de l'exécution, à la manière des continuations. D'autre part elle donne une vision abstraite des programmes en se focalisant uniquement sur les phénomènes récursifs. Enfin, elle permet d'exprimer un moyen d'obfusquer les programmes.

Chapitre 3

Machine réflexive

Dans ce chapitre on va présenter une machine où les comportements réflexifs sont mis en avant.

Dans les machines de Turing ou les RAM (Random Access Machines) [CR73], le programme, isolé des données, peut lire et modifier des données. Mais lui même ne peut être lu ou modifié. Les RASP [ER64 ; Har71] (Random Access Stored Program), au contraire, confondent programme et données : le programme est logé en mémoire et peut à loisir être modifié. Si $\mathcal{A}$ est un ensemble d'adresses, la *mémoire* d'une machine RASP est une fonction $h : \mathcal{A} \to \mathbb{N}$. Un *état* s de la machine est donné par le couple (h, a, n) où h est une mémoire, a est une adresse et n un entier. L'ensemble des entiers $\mathbb{N}$ est utilisé pour encoder les *instructions* de la machine. L'ensemble de ces instructions est noté $\mathcal{I}$. On dispose donc d'une *fonction d'encodage* de $\mathcal{I} \to \mathbb{N}$. Si $i \in \mathcal{I}$ est une instruction, on note $\overline{i} \in \mathbb{N}$ cet encodage. La sémantique opérationnelle de la machine est donnée par les règles $s_1 \xrightarrow{i} s_2$, où, si $s_1 = (h_1, a_1, n_1)$, $\overline{i} = h_1(a_1)$. Ces règles sont données dans la Figure 3.1. La machine s'arrête quand $h(a)$ ne correspond à l'encodage d'aucune instruction. Cependant, ces machines manquent de structures pour parler de réflexion, c'est à dire le rapport qu'il y a entre données et programmes. En effet, la machine ne fait pas de distinction entre les données et les programmes.

Les TRM (Text Register Machines) [Mos06] apportent une structure plus simple à manipuler et les SRM (Self-modifying Register Machine) de [Mar12] ajoutent le coté reflexif. Cependant, la distinction donnée/programme n'est toujours pas mis en avant, contrairement à notre modèle de machine : alors que données et programmes sont indistinguables dans les SRM, on insiste sur cette distinction en séparant clairement les programmes des données d'une part, mais on permet le passage d'un état à un autre en le spécifiant explicitement.

3.1 Machines auto-modifiantes à piles de registres

3.1.1 Présentation

Les SRM sont composées d'une pile de registres, avec à son sommet le registre contenant le programme à exécuter. On propose une variation de ce modèle avec les SSRM

$$
\begin{array}{ll}
s = (h, a, n) & \\
\xrightarrow{\text{TRA } a'} & (h, a', n) \\
\xrightarrow{\text{TRA } \langle a' \rangle} & (h, h(a'), n) \\
\xrightarrow{\text{TRZ } a'} & (h, a', n) \text{ si } n = 0, (h, a+1, n) \text{ sinon} \\
\xrightarrow{\text{TRZ } \langle a' \rangle} & (h, h(a'), n) \text{ si } n = 0, (h, a+1, n) \text{ sinon} \\
\xrightarrow{\text{STO } a'} & (h[a' \leftarrow n], a+1, n) \\
\xrightarrow{\text{STO } \langle a' \rangle} & (h[h(a') \leftarrow n], a+1, n) \\
\xrightarrow{\text{CLA } n'} & (h, a+1, n') \\
\xrightarrow{\text{CLA } \langle a' \rangle} & (h, a+1, h(a')) \\
\xrightarrow{\text{CLA } \langle \langle a' \rangle \rangle} & (h, a+1, h(h(a'))) \\
\xrightarrow{\text{ADD } n'} & (h, a+1, n+n') \\
\xrightarrow{\text{ADD } \langle a' \rangle} & (h, a+1, n+h(a')) \\
\xrightarrow{\text{ADD } \langle \langle a' \rangle \rangle} & (h, a+1, n+h(h(a'))) \\
\xrightarrow{\text{SUB } n'} & (h, a+1, n-n') \\
\xrightarrow{\text{SUB } \langle a' \rangle} & (h, a+1, n-h(a')) \\
\xrightarrow{\text{SUB } \langle \langle a' \rangle \rangle} & (h, a+1, n-h(h(a')))
\end{array}
$$

FIGURE 3.1 – Sémantique opérationnelle de la RASP

(Stack Self-modifying Register Machine), qui, elles, comportent *deux* piles, appelées *zones*, une pour les données (**d**), et une autre pour les programmes (**x**).

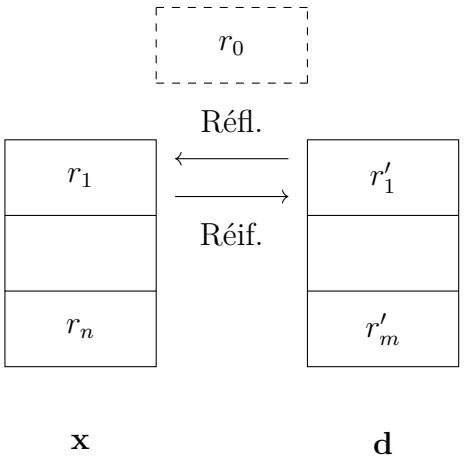
Le registre r en tête de **x** (on note $\mathbf{x} = r.\mathbf{x}'$) contient le programme qui s'exécute à l'instant courant. Un pointeur d'instruction a indique quelle instruction exécuter dans ce registre. Lorsque l'exécution de ce registre est finie, ce registre est détruit, et on exécute le registre suivant dans la pile **x**. La zone **x** ressemble, en ce sens, à une pile d'exécution.

La zone **d** est la pile des données. De la même manière que pour **x**, seul le registre en tête de **d** peut être lu ou modifié. Cependant, on peut effectuer une permutation circulaire sur **d**, à gauche et à droite, ce qui permet d'accéder à n'importe quel registre de **d**.

Chaque registre est un mot sur l'alphabet $\{\mathtt{1}, \#\}$ (à la manière de [Mos06 ; Mar12]). On y accède à la manière d'une file d'attente (FIFO) : l'accès en lecture consomme le premier caractère du mot, et l'ajout d'une lettre se fait à la fin du mot.

La particularité intéressante de notre machine est que les registres des zones **x** et **d** peuvent bouger d'une zone à l'autre. On *désactive* un registre lorsqu'on le fait passer de la zone **x** à la zone **d**, on l'*active* lorsqu'on fait l'inverse. Il est à noter que les échanges se font entre le sommet de **d** et le registre juste après le sommet de **x**. En effet, le sommet de **x** est le programme en train de s'exécuter, il est intouchable.

Pour simplifier, étant donné le rôle à part de la tête de la zone **x** (ce registre n'est pas désactivable et les activations se font juste après lui), on peut la considérer comme séparée de **x**. De cette manière, les désactivations et les activations se font aussi au sommet de **x**. Si $\mathbf{x} = [r_0, r_1, \cdots, r_n]$ et $\mathbf{d} = [r'_1, \cdots, r'_m]$, on peut représenter la machine de la manière suivante.



Le passage d'un registre d'une zone à une autre est explicite (une instruction pour chaque sens), contrairement aux SRM ou aux RASP. On change de paradigme : alors que la réflexion est toujours permise, elle est encadrée.

3.1.2 Structure de la machine

Un *état* $s = (\mathbf{x}, a, \mathbf{d})$ de la machine est composée de deux *zones* **x** et **d**. Chacune de ces zones est une pile de *registres*. Un registre est un mot sur l'alphabet $\{\mathtt{1}, \#\}$. L'état

contient aussi un entier a, le *pointeur d'instruction*.

$$\begin{array}{rcll} a & \in & \mathcal{A} \stackrel{\text{def}}{=} \mathbb{N} & \text{(pointeur d'instruction)} \\ r, p, d & \in & \mathcal{R} \stackrel{\text{def}}{=} \{\mathtt{1}, \#\}^* & \text{(registres)} \\ \mathbf{x}, \mathbf{d} & \in & \mathcal{Z} \stackrel{\text{def}}{=} \mathcal{R}^* & \text{(zones)} \\ s & \in & \mathcal{S} \stackrel{\text{def}}{=} \mathcal{Z} \times \mathcal{A} \times \mathcal{Z} & \text{(états)} \end{array}$$

Dans la suite, la lettre r désignera un registre quelconque (dans $\mathbf{x}$ ou $\mathbf{d}$). Lorsqu'on voudra plus spécifiquement parler d'un registre dans $\mathbf{x}$, on notera plutôt p, et d lorsqu'on voudra parler d'un registre de $\mathbf{d}$.

La machine est contrôlée par des *instructions* encodées dans les mots de l'alphabet $\{\mathtt{1}, \#\}$. L'encodage choisi est donné dans la Figure 3.2. Si i est une instruction, on notera $\overline{i}$ son codage. On pourra définir un registre à partir des instructions qu'il contient. Par exemple, le registre $\overline{\mathtt{inact}} \bullet \overline{\mathtt{act}} = \mathtt{1}\#^9\mathtt{1}\#^8$ pourra être noté

$$\left\{ \begin{array}{l} \mathtt{inact} \\ \mathtt{act} \end{array} \right\}$$

On peut incorporer le code d'un autre programme p dans cette représentation. L'apparition d'un programme aux cotés des instructions est notée $\underline{p}$. Par exemple, le programme $\mathtt{1}\#^9 \bullet p \bullet \mathtt{1}\#^8$ pourra être noté

$$\left\{ \begin{array}{l} \mathtt{inact} \\ \underline{p} \\ \mathtt{act} \end{array} \right\}$$

Enfin, pour plus de concision, on pourra utiliser des sauts sur des *labels* plutôt que de donner le nombre d'instructions à passer. Par exemple

$$\left\{ \begin{array}{ll} @_0 & \mathtt{inact} \\ & \underline{p} \\ & \mathtt{jump}\, @_0 \\ & \mathtt{act} \end{array} \right\} = \left\{ \begin{array}{l} \mathtt{inact} \\ \underline{p} \\ \mathtt{jump}\, -(|p|+1) \\ \mathtt{act} \end{array} \right\}$$

On utilisera également le label spécial $@_{\mathtt{exit}}$ désignant la sortie du programme. C'est à dire que l'instruction $\mathtt{jump}\, @_{\mathtt{exit}}$ saute juste après la dernière instruction du bloc courant.

Dans la suite on utilisera les notations suivantes

- La longueur d'une zone $\mathbf{z}$ ($\mathbf{x}$ ou $\mathbf{d}$) sera notée $|\mathbf{z}|$.
- On note $r.\mathbf{z}$ la zone obtenue en ajoutant r au somment de $\mathbf{z}$. De même, si $e \in \{\mathtt{1}, \#\}$, on note $e.r$ le registre obtenu en ajoutant e au début de r et $r.e$ l'ajoute de e à la fin de r. Le registre vide sera noté ϵ.
- Si r_1 et r_2 sont deux registres, on note $r_1 \bullet r_2$ le registre obtenu en les concaténant.
- Si $\mathbf{d} = [r_1, \cdots, r_n]$ est une zone, on note $\overrightarrow{\mathbf{d}}$ le shift droit de $\mathbf{d}$, c'est à dire $\overrightarrow{\mathbf{d}} \stackrel{\text{def}}{=} [r_n, r_1, \cdots, r_{n-1}]$. De même, on note $\overleftarrow{\mathbf{d}}$ le shift gauche de $\mathbf{d}$, c'est à dire $\overleftarrow{\mathbf{d}} \stackrel{\text{def}}{=} [r_2, \cdots, r_n, r_1]$.
- On notera $r[a]$ la a-ème instruction du registre r. On insistera sur le fait qu'il ne s'agit pas du a-ème caractère de r. Ici, on compte en nombre d'instructions, et pas en nombre de caractères.
- Si $\mathbf{x} = r.\mathbf{x}'$, on note $\mathbf{x}[a]$ l'instruction $r[a]$.

Instruction	Codage	Explication
`put #`	$\texttt{1}\#^1$	ajoute # à la fin du premier registre de **d**
`put 1`	$\texttt{1}\#^2$	ajoute `1` à la fin du premier registre de **d**
`new`	$\texttt{1}\#^3$	ajoute un registre vide au sommet de **d**
`right`	$\texttt{1}\#^4$	effectue une permutation circulaire des registres de **d** vers la droite
`left`	$\texttt{1}\#^5$	effectue une permutation circulaire des registres de **d** vers la gauche
`case`	$\texttt{1}\#^6$	branche en fonction de la première lettre du premier registre de **d**
`pop`	$\texttt{1}\#^7$	supprime le premier registre de **d**
`act`	$\texttt{1}\#^8$	active le premier registre de **d**
`inact`	$\texttt{1}\#^9$	désactive le deuxième registre de **x**
`jump` $+n$	$\texttt{1}^{n+1}\#^{10}$	saute de n instructions en avant dans le registre
`jump` $-n$	$\texttt{1}^{n+1}\#^{11}$	saute de n instructions en arrière dans le registre

FIGURE 3.2 – Encodage des instructions

3.1.3 Opérations de la machine

3.1.3.1 Explication des instructions

Chaque registre est vu comme une file, c'est à dire qu'il est possible d'ajouter une lettre # ou `1` à fin de la file (`put`), et que son premier élément peut être supprimé lors de sa lecture (`case`). Les deux instructions ne peuvent accéder qu'au premier registre de la pile **d**. Cependant, il est possible de faire tourner la pile avec les instructions `right` et `left`, ce qui permet d'accéder à n'importe quel registre de la pile.

À un niveau supérieur, la zone **d** est une pile. Il est possible d'ajouter un registre vide à son sommet (`new`) ou de supprimer le registre au sommet (`pop`).

Le registre r contenant le programme à exécuter se trouve au sommet de $\mathbf{x} = r.\mathbf{x}'$. L'instruction à exécuter dans ce registre est donnée par le pointer d'instruction a. On rappelle ici que a ne désigne pas le a-ème caractère du registre, mais bien la a-ème instruction, c'est à dire $r[a]$. Le contrôle de l'exécution est permis grâce aux instructions de saut (`jump`). Les sauts s'effectuent relativement au pointeur a, et donnent également le nombre d'instructions à sauter, et pas le nombre de caractères. Lorsque le pointeur d'instruction est égal à la taille du registre (c'est à dire qu'il pointe sur le "vide" juste après le registre), on supprime le registre au sommet de **x** et on passe à la première instruction du registre suivant, s'il existe. Dans le cas contraire, le calcul s'arrête.

Il est enfin possible de faire passer le registre au sommet de **d** sous le sommet de **x** (le sommet de **x** étant le registre exécuté, on introduit le nouveau registre juste en dessous de lui) avec l'instruction `act`. L'instruction `inact` fait exactement l'inverse, c'est à dire qu'elle fait passer le registre sous le sommet de **x** au sommet de **d**.

3.1.3.2 Localité du calcul

La machine est conçue de manière à ce que les opérations sur les registres se passent "près du sommet" de **d** et **x**. Ces zones disposent des opérations usuelles sur les piles (on peut ajouter et supprimer des éléments à leur sommet). De plus, les opérations de

$$
\begin{array}{lrcl}
\mathbf{x}[a] = \texttt{put}\,\# & \langle \mathbf{x}, a, r.\mathbf{d}\rangle & \to & \langle \mathbf{x}, a+1, (r.\#).\mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{put}\,1 & \langle \mathbf{x}, a, r.\mathbf{d}\rangle & \to & \langle \mathbf{x}, a+1, (r.1).\mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{right} & \langle \mathbf{x}, a, \mathbf{d}\rangle & \to & \langle \mathbf{x}, a+1, \overrightarrow{\mathbf{d}}\rangle \\
\mathbf{x}[a] = \texttt{left} & \langle \mathbf{x}, a, \mathbf{d}\rangle & \to & \langle \mathbf{x}, a+1, \overleftarrow{\mathbf{d}}\rangle \\
\mathbf{x}[a] = \texttt{jump}\,{+n} & \langle \mathbf{x}, a, \mathbf{d}\rangle & \to & \langle \mathbf{x}, a+n, \mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{jump}\,{-n} & \langle \mathbf{x}, a, \mathbf{d}\rangle & \to & \langle \mathbf{x}, a-n, \mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{case} & \langle \mathbf{x}, a, (\#.r).\mathbf{d}\rangle & \to & \langle \mathbf{x}, a+1, r.\mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{case} & \langle \mathbf{x}, a, (1.r).\mathbf{d}\rangle & \to & \langle \mathbf{x}, a+2, r.\mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{case} & \langle \mathbf{x}, a, \epsilon.\mathbf{d}\rangle & \to & \langle \mathbf{x}, a+3, \epsilon.\mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{pop} & \langle \mathbf{x}, a, r.\mathbf{d}\rangle & \to & \langle \mathbf{x}, a+1, \mathbf{d}\rangle \\
\mathbf{x}[a] = \texttt{new} & \langle \mathbf{x}, a, \mathbf{d}\rangle & \to & \langle \mathbf{x}, a+1, \epsilon.\mathbf{d}\rangle \\
r_{\mathbf{x}}[a] = \texttt{act} & \langle r_{\mathbf{x}}.\mathbf{x}, a, r_{\mathbf{d}}.\mathbf{d}\rangle & \to & \langle r_{\mathbf{x}}.r_{\mathbf{d}}.\mathbf{x}, a+1, \mathbf{d}\rangle \\
r_{\mathbf{x}}[a] = \texttt{inact} & \langle r_{\mathbf{x}}.r'_{\mathbf{x}}.\mathbf{x}, a, \mathbf{d}\rangle & \to & \langle r_{\mathbf{x}}.\mathbf{x}, a+1, r'_{\mathbf{x}}.\mathbf{d}\rangle \\
\hline
 & \langle r.\mathbf{x}, |r|, \mathbf{d}\rangle & \to & \langle \mathbf{x}, 0, \mathbf{d}\rangle
\end{array}
$$

FIGURE 3.3 – Transitions entre les états

modification des registres de $\mathbf{d}$ (`put` et `case`) ne peuvent modifier que les registres en sommet de pile. De même les instructions de rotation de $\mathbf{d}$ sont limitées à une seule rotation.

3.1.3.3 Transitions entre états

La sémantique opérationnelle de la machine est donnée par la relation de transition $\to \in \wp(\mathcal{S}^2)$. On a $\langle \mathbf{x}, a, \mathbf{d}\rangle \to \langle \mathbf{x}', a', \mathbf{d}'\rangle$ lorsque la transformation correspond à l'instruction pointée par a dans le registre au sommet de $\mathbf{x}$. On rappelle que cette instruction est notée $\mathbf{x}[a]$. La relation est donnée dans la Figure 3.3. On notera que la dernière transition ne correspond pas à une instruction. Elle est déclenchée automatiquement lorsque le pointeur d'instruction désigne le "vide" juste après le mot du registre. On note $\xrightarrow{*}$ la clôture réflexive transitive de $\to$. On notera $s \not\to$ lorsque s ne se réduit par $\to$. Plusieurs cas d'arrêt sont possibles.

— La zone $\mathbf{x}$ est vide et $a = 0$.

— Le pointeur a est strictement supérieur à $|\mathbf{x}[0]|$.

— Le mot pointé pas a ne correspond par à une instruction.

— Les accès (`put`, `pop`, `case`, `act`, `inact`) à la tête de $\mathbf{d}$ ou au deuxième élément de $\mathbf{x}$ échouent.

Seul le premier cas est un arrêt normal de la machine. Tous les autres sont considérés comme des erreurs à l'exécution.

3.1.3.4 Représentation des calculs

On donne une autre représentation des opérations dans notre machine. On note $\langle r_n, \cdots, r_1 \mid r_0 \bullet \boxed{r'_0} \mid r'_1, \cdots, r'_m\rangle$ l'état $\langle [r_0 \bullet r'_0, r_1, \cdots, r_n], a, [r'_1, \cdots, r'_m]\rangle$, et où a indique la première instruction de r'_0. Cette notation permet de mettre en évidence les

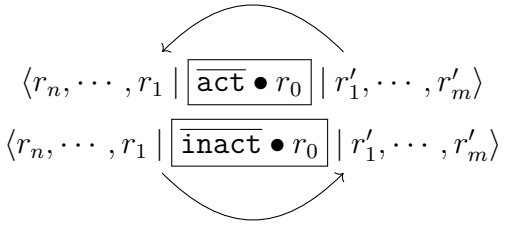
mouvements de registre lors du calcul de la machine. Par exemple,

$$\langle r_n, \cdots, r_1 \mid \boxed{\overline{\texttt{act}} \bullet r_0} \mid r'_1, \cdots, r'_m \rangle \to \langle r_n, \cdots, r_1, r'_1 \mid \overline{\texttt{act}} \bullet \boxed{r_0} \mid r'_2, \cdots, r'_m \rangle$$

la donnée r'_1 dans $\mathbf{d}$ devient exécutable en passant dans la zone des programmes $\mathbf{x}$ et la machine continue avec l'exécution de r_0. De même,

$$\langle r_n, \cdots, r_1 \mid \boxed{\overline{\texttt{inact}} \bullet r_0} \mid r'_1, \cdots, r'_m \rangle \to \langle r_n, \cdots, r_2 \mid \overline{\texttt{inact}} \bullet \boxed{r_0} \mid r_1, r'_1, \cdots, r'_m \rangle$$

le registre exécutable r_1 dans $\mathbf{x}$ devient une donnée en passant dans la zone des données $\mathbf{d}$. On notera la symétrie de ces opérations.



On note

$$\langle r_n, \cdots, r_1 \mid r_0 \mid r'_1, \cdots, r'_m \rangle$$

lorsque $a = |r_0|$. Dans ce cas

$$\langle r_n, \cdots, r_1 \mid r_0 \mid r'_1, \cdots, r'_m \rangle \to \langle r_n, \cdots, r_2 \mid \boxed{r_1} \mid r'_1, \cdots, r'_m \rangle$$

3.1.3.5 Fonction calculée par la machine

La fonction calculée par un programme $p \in \{\mathtt{1}, \#\}^*$, notée $[\![p]\!] : \mathcal{D}^* \to \mathcal{D}^*$ est définie par

$$\begin{array}{rcll} [\![p]\!](\mathbf{d}) & \stackrel{\text{def}}{=} & \mathbf{d}' & \text{lorsque } \langle [p], 0, \mathbf{d} \rangle \stackrel{*}{\to} \langle \epsilon, 0, \mathbf{d}' \rangle \\ & \stackrel{\text{def}}{=} & \text{non défini} & \text{sinon} \end{array}$$

Les programmes et les données sont représentés par des mots dans les registres. La différence entre programmes et données est donc uniquement structurelle : elle dépend de la position du registre ($\mathbf{x}$ ou $\mathbf{d}$) dans la machine.

3.2 Utilisation

3.2.1 Programmes de base

3.2.1.1 Machine universelle

On dispose d'une machine universelle univ

$$\langle p_n, \cdots, p_1 \mid \boxed{\mathsf{univ}} \mid d_1, \cdots, d_m \rangle \stackrel{*}{\to} \langle p_n, \cdots, p_1 \mid \boxed{d_1} \mid d_2, \cdots, d_m \rangle$$

On peut écrire cette machine universelle de la manière suivante

$$\mathsf{univ} \stackrel{\text{def}}{=} \overline{\texttt{act}}$$

On a donc

$$[\![\mathsf{univ}]\!](p.\mathbf{d}) = [\![p]\!](\mathbf{d})$$

3.2.1.2 Accès à distance

Grâce aux instructions de rotation `right` et `left`, on peut accéder au registre à distance n du sommet dans $\mathbf{d}$. Pour cela on peut déjà définir l'opération $\mathbf{right}(i)$ et $\mathbf{left}(i)$ qui itèrent i fois les instructions `right` et `left`.

$$\begin{array}{rcl} \mathbf{right}(i) & \stackrel{\text{def}}{=} & \overline{\texttt{right}}^{i} \\ \mathbf{left}(i) & \stackrel{\text{def}}{=} & \overline{\texttt{left}}^{i} \end{array}$$

On peut alors définir les programmes

$$\begin{array}{rcl} \mathbf{put}(i,\texttt{1}) & \stackrel{\text{def}}{=} & \mathbf{left}(i) \bullet \overline{\texttt{put 1}} \bullet \mathbf{right}(i) \\ \mathbf{put}(i,\#) & \stackrel{\text{def}}{=} & \mathbf{left}(i) \bullet \overline{\texttt{put \#}} \bullet \mathbf{right}(i) \end{array}$$

De même on peut définir le programme

$$\mathbf{case}(i) \stackrel{\text{def}}{=} \left\{ \begin{array}{lll} & \underline{\mathbf{left}(i)} & \textit{accès au } i\textit{-ème registre de } \mathbf{d} \\ & \texttt{case} & \\ & \texttt{jump}\,@_1 & \\ & \texttt{jump}\,@_2 & \\ & \texttt{jump}\,@_3 & \\ @_1 & \underline{\mathbf{right}(i)} & \textit{restauration de } \mathbf{d} \\ & \texttt{jump}\,@_{\texttt{exit}} & \textit{saute sur la 1-ère instruction après le programme} \\ @_2 & \underline{\mathbf{right}(i)} & \textit{restauration de } \mathbf{d} \\ & \texttt{jump}\,@_{\texttt{exit}}+1 & \textit{saute sur la 2-ème instruction après le programme} \\ @_3 & \underline{\mathbf{right}(i)} & \textit{restauration de } \mathbf{d} \\ & \texttt{jump}\,@_{\texttt{exit}}+2 & \textit{saute sur la 3-ème instruction après le programme} \end{array} \right\}$$

3.2.1.3 Déplacement de registre

On peut écrire le programme $\mathbf{move}(i,j)$ déplaçant le registre i à la place j

$$\begin{array}{rl} & \langle p_n, \cdots, p_1 \mid \boxed{\mathbf{move}(i,j) \bullet p_0} \mid d_0, \cdots, d_i, \cdots, d_j, \cdots, d_m \rangle \\ \stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathbf{move}(i,j) \bullet \boxed{p_0} \mid d_0, \cdots, \epsilon, \cdots, d_j \bullet d_i, \cdots, d_m \rangle \end{array}$$

On peut l'écrire

$$\mathbf{move}(i,j) \stackrel{\text{def}}{=} \left\{ \begin{array}{lll} @_0 & \underline{\mathbf{case}(i)} & \textit{lecture du premier caractère du } i\textit{-ème registre} \\ & \texttt{jump}\,@_1 & \textit{si } d_i = \#.d_i' \\ & \texttt{jump}\,@_2 & \textit{si } d_i = \texttt{1}.d_i' \\ & \texttt{jump}\,@_{\texttt{exit}} & \textit{si } d_i = \epsilon \textit{ (on s'arrête)} \\ @_1 & \underline{\mathbf{put}(j,\#)} & d_i = \#.d_i' \\ & \texttt{jump}\,@_0 & \\ @_2 & \underline{\mathbf{put}(j,\texttt{1})} & d_i = \texttt{1}.d_i' \\ & \texttt{jump}\,@_0 & \end{array} \right\}$$

On a donc

$$[\![\mathbf{move}(i,j)]\!]\,([d_1, \cdots, d_i, \cdots, d_j, \cdots, d_m]) = [d_1, \cdots, \epsilon, \cdots, d_j \bullet d_i, \cdots, d_m]$$

3.2.1.4 Duplication

Le programme dupl duplique son entrée

$$\langle p_n, \cdots, p_1 \mid \boxed{\mathsf{dupl} \bullet p_0} \mid d_1, \cdots, d_m \rangle \xrightarrow{*} \langle p_n, \cdots, p_1 \mid \mathsf{dupl} \bullet \boxed{p_0} \mid d_1, d_1, \cdots, d_m \rangle$$

Il peut être écrit de la manière suivante

$$\mathsf{dupl} \stackrel{\text{def}}{=} \left\{ \begin{array}{lll} & \texttt{new} & \textit{création de deux nouveaux registres} \\ & \texttt{new} & \\ @_0 & \underline{\mathbf{case}(2)} & \textit{lecture de } d \\ & \underline{\texttt{jump}\, @_1} & \textit{si } d = \#.d' \\ & \texttt{jump}\, @_2 & \textit{si } d = \texttt{1}.d' \\ & \texttt{jump}\, @_3 & \textit{si } d = \epsilon \\ @_1 & \underline{\mathbf{put}(0, \#)} & d = \#.d' \\ & \underline{\mathbf{put}(1, \#)} & \\ & \underline{\texttt{jump}\, @_0} & \\ @_2 & \underline{\mathbf{put}(0, \texttt{1})} & d = \texttt{1}.d' \\ & \underline{\mathbf{put}(1, \texttt{1})} & \\ & \underline{\texttt{jump}\, @_0} & \\ @_3 & \underline{\mathbf{move}(1, 2)} & \textit{décalage des deux registres en tête} \\ & \underline{\mathbf{move}(0, 1)} & \\ & \texttt{pop} & \textit{suppression du registre (vide) en tête} \end{array} \right\}$$

On a donc

$$[\![\mathsf{dupl}]\!]\,(d.\mathbf{d}) = d.d\mathbf{d}$$

3.2.1.5 Commutateur

Le programme switch défini par

$$\langle p_n, \cdots, p_1 \mid \boxed{\mathsf{switch} \bullet p_0} \mid d_1, d_2, \cdots, d_m \rangle \xrightarrow{*} \langle p_n, \cdots, p_1 \mid \mathsf{switch} \bullet \boxed{p_0} \mid d_2, d_1, \cdots, d_m \rangle$$

peut être défini par

$$\mathsf{switch} \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \texttt{new} & \textit{création d'un nouveau registre} \\ \underline{\mathbf{move}(2, 0)} & \\ \underline{\mathbf{move}(1, 2)} & \\ \underline{\mathbf{move}(0, 1)} & \end{array} \right\}$$

On a donc

$$[\![\mathsf{switch}]\!]\,(d_1.d_2.\mathbf{d}) = d_2.d_1.\mathbf{d}$$

3.2.2 Reproduction

3.2.2.1 Fonction d'écriture

La fonction d'écriture **put** (on utilise la même notation que la généralisation de `put`) vérifie

$$\langle p_n, \cdots, p_1 \mid \boxed{\mathbf{put}(d) \bullet p_0} \mid d_1, \cdots, d_m \rangle \xrightarrow{*} \langle p_n, \cdots, p_1 \mid \mathbf{write}(d) \bullet \boxed{p_0} \mid d_1 \bullet d, \cdots, d_m \rangle$$

Si $d = [e_0, \cdots, e_n]$, où $e_i \in \{0, 1\}$, alors on peut écrire $\mathbf{put}(d)$ de la manière suivante :

$$\mathbf{put}(d) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{put}\, e_0 \\ \cdots \\ \texttt{put}\, e_n \end{array} \right\}$$

3.2.2.2 Fonction de reproduction

La fonction de reproduction **write** génère un programme $\mathbf{write}(d)$ qui écrit la donnée d. Il vérifie

$$\langle p_n, \cdots, p_1 \mid \boxed{\mathbf{write}(d) \bullet p_0} \mid d_1, \cdots, d_m \rangle \stackrel{*}{\rightarrow} \langle p_n, \cdots, p_1 \mid \mathbf{write}(d) \bullet \boxed{p_0} \mid d, d_1, \cdots, d_m \rangle$$

On peut définir **write** de la manière suivante

$$\mathbf{write} \stackrel{\text{def}}{=} \overline{\texttt{new}} \bullet \mathbf{put}(d)$$

3.2.2.3 Reproducteur

Le programme write implémentant la fonction **write** vérifie

$$\langle p_n, \cdots, p_1 \mid \boxed{\mathsf{write} \bullet p_0} \mid d_1, \cdots, d_m \rangle \stackrel{*}{\rightarrow} \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \boxed{p_0} \mid \mathbf{write}(d_1), \cdots, d_m \rangle$$

et on a

$$[\![\mathsf{write}]\!]\,(d.\mathbf{d}) = \mathbf{write}(d).\mathbf{d}$$

et peut être donné par

$$\mathsf{write} \stackrel{\text{def}}{=} \left\{ \begin{array}{lll} & \texttt{new} & \\ & \underline{\mathbf{put}(\overline{\texttt{new}})} & \textit{écrit le code de } \texttt{new} \\ @_0 & \underline{\mathbf{case}(1)} & \\ & \texttt{jump}\, @_1 & \textit{si } d = \#.d' \\ & \texttt{jump}\, @_2 & \textit{si } d = \texttt{1}.d' \\ & \texttt{jump}\, @_3 & \textit{si } d = \epsilon \\ @_1 & \underline{\mathbf{put}(\overline{\texttt{put}\ \#})} & \textit{écrit le code de } \texttt{put}\ \# \\ & \underline{\texttt{jump}\, @_0} & \\ @_2 & \underline{\mathbf{put}(\overline{\texttt{put 1}})} & \textit{écrit le code de } \texttt{put 1} \\ & \underline{\texttt{jump}\, @_0} & \\ @_3 & \underline{\mathbf{move}(0, 1)} & \\ & \texttt{pop} & \end{array} \right\}$$

3.2.3 Spécialisation

3.2.3.1 Fonction de spécialisation

La fonction de spécialisation **spec** est donnée par

$$[\![\mathbf{spec}(p, d)]\!]\,(\mathbf{d}) = [\![p]\!]\,(d.\mathbf{d})$$

Elle peut être définie par

$$\mathbf{spec}(p,d) \stackrel{\text{def}}{=} \mathbf{write}(d) \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}$$

On a en effet

$$\begin{array}{rl}
 & \langle p_n, \cdots, p_1 \mid \boxed{\mathbf{write}(d) \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}} \mid d_1, \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathbf{write}(d) \bullet \boxed{\mathbf{write}(p) \bullet \overline{\mathtt{act}}} \mid d, d_1, \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathbf{write}(d) \bullet \mathbf{write}(p) \bullet \boxed{\overline{\mathtt{act}}} \mid p, d, d_1, \cdots, d_m \rangle \\
\rightarrow & \langle p_n, \cdots, p_1, p \mid \mathbf{write}(d) \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}} \mid d, d_1, \cdots, d_m \rangle \\
\rightarrow & \langle p_n, \cdots, p_1 \mid \boxed{p} \mid d, d_1, \cdots, d_m \rangle
\end{array}$$

3.2.3.2 Spécialiseur

Le programme spec implémente **spec**, c'est à dire que

$$[\![\mathsf{spec}]\!]\,(p.d.\mathbf{d}) = \mathbf{spec}(p,d).\mathbf{d}$$

On peut écrire

$$\mathsf{spec} \stackrel{\text{def}}{=} \mathsf{write} \bullet \mathsf{switch} \bullet \mathsf{write} \bullet \mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \overline{\mathtt{pop}}$$

On a en effet

$$\begin{array}{rl}
 & \langle p_n, \cdots, p_1 \mid \boxed{\mathsf{write} \bullet \mathsf{switch} \bullet \mathsf{write} \bullet \mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \overline{\mathtt{pop}} \bullet p_0} \mid p, d_1, \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \boxed{\mathsf{switch} \bullet \mathsf{write} \bullet \mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \overline{\mathtt{pop}} \bullet p_0} \mid \mathbf{write}(p), d_1, \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \mathsf{switch} \bullet \boxed{\mathsf{write} \bullet \mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \overline{\mathtt{pop}} \bullet p_0} \mid d_1, \mathbf{write}(p), \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \mathsf{switch} \bullet \mathsf{write} \bullet \boxed{\mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \overline{\mathtt{pop}} \bullet p_0} \mid \mathbf{write}(d_1), \mathbf{write}(p), \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \mathsf{switch} \bullet \mathsf{write} \bullet \mathbf{move}(1,0) \bullet \boxed{\mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \overline{\mathtt{pop}} \bullet p_0} \mid \mathbf{write}(d_1) \bullet \mathbf{write}(p), \epsilon, \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \mathsf{switch} \bullet \mathsf{write} \bullet \mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \boxed{\mathsf{switch} \bullet \overline{\mathtt{pop}} \bullet p_0} \mid \mathbf{write}(d_1) \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}, \epsilon, \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \mathsf{switch} \bullet \mathsf{write} \bullet \mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \boxed{\overline{\mathtt{pop}} \bullet p_0} \mid \epsilon, \mathbf{write}(d_1) \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}, \cdots, d_m \rangle \\
\stackrel{*}{\rightarrow} & \langle p_n, \cdots, p_1 \mid \mathsf{write} \bullet \mathsf{switch} \bullet \mathsf{write} \bullet \mathbf{move}(1,0) \bullet \mathbf{put}(\overline{\mathtt{act}}) \bullet \mathsf{switch} \bullet \overline{\mathtt{pop}} \bullet \boxed{p_0} \mid \mathbf{write}(d_1) \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}, \cdots, d_m \rangle
\end{array}$$

3.2.4 Point fixe

3.2.4.1 Auto-application

On pose omega le programme tel que

$$\langle p_n, \cdots, p_1 \mid \boxed{\mathsf{omega} \bullet p_0} \mid d_1, \cdots, d_m \rangle \stackrel{*}{\rightarrow} \langle p_n, \cdots, p_1 \mid \mathsf{omega} \bullet \boxed{p_0} \mid \mathbf{spec}(d_1, d_1), \cdots, d_m \rangle$$

c'est à dire

$$[\![\mathsf{omega}]\!]\,(d_1.\mathbf{d}) = \mathbf{spec}(d_1, d_1).\mathbf{d}$$

que l'on peut écrire de la manière suivante

$$\mathsf{omega} \stackrel{\text{def}}{=} \mathsf{dupl} \bullet \mathsf{spec}$$

3.2.4.2 Point fixe

On cherche le programme $\mathbf{fix}(p)$ tel que

$$[\![\mathbf{fix}(p)]\!]\,(\mathbf{d}) = [\![p]\!]\,(\mathbf{fix}(p).\mathbf{d})$$

Il peut être donné par

$$\mathbf{fix}(p) \stackrel{\text{def}}{=} \mathbf{spec}(\mathsf{omega} \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}, \mathsf{omega} \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}})$$

On a en effet

$$\begin{array}{ll}
 & \langle p_n, \cdots, p_1 \mid \boxed{\mathbf{fix}(p)} \mid d_1, \cdots, d_m \rangle \\
\stackrel{*}{\to} & \langle p_n, \cdots, p_1 \mid \boxed{\mathsf{omega} \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}} \mid \mathsf{omega} \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}}, d_1, \cdots, d_m \rangle \\
\stackrel{*}{\to} & \langle p_n, \cdots, p_1 \mid \mathsf{omega} \bullet \boxed{\mathbf{write}(p) \bullet \overline{\mathtt{act}}} \mid \mathbf{fix}(p), d_1, \cdots, d_m \rangle \\
\stackrel{*}{\to} & \langle p_n, \cdots, p_1 \mid \mathsf{omega} \bullet \mathbf{write}(p) \bullet \boxed{\overline{\mathtt{act}}} \mid p, \mathbf{fix}(p), d_1, \cdots, d_m \rangle \\
\to & \langle p_n, \cdots, p_1, p \mid \mathsf{omega} \bullet \mathbf{write}(p) \bullet \overline{\mathtt{act}} \mid \mathbf{fix}(p), d_1, \cdots, d_m \rangle \\
\to & \langle p_n, \cdots, p_1 \mid \boxed{p} \mid \mathbf{fix}(p), d_1, \cdots, d_m \rangle
\end{array}$$

3.3 Sémantique concrète

Dans cette section, on va introduire la notion intentionnelle de *trace*, qui donne une information sur le "chemin" pris par le calcul.

Définition 3.3.1 (Chemin)**.** Un *chemin* σ est un élément de $\mathcal{S}^*$, où on rappelle que $\mathcal{S}$ est l'ensemble des états s de la machine. Si σ est le chemin $[s_0, \cdots, s_n]$ et s un état, on notera $s.\sigma$ le chemin $[s, s_0, \cdots, s_n]$ et $\sigma.s$ le chemin $[s_0, \cdots, s_n, s]$.

Un chemin est une liste d'états $s \in \mathcal{S}$ par lesquels passe le calcul. Si σ est un chemin, on rappelle la notation

$$\mathrm{pref}(\sigma) = \{\sigma' \mid \exists \sigma'', \sigma = \sigma' \bullet \sigma''\}$$

L'ensemble $\mathrm{pref}(\sigma)$ contient tous les sous-chemins commençant σ.

Un chemin est une séquence finie des étapes d'un calcul. L'ensemble des chemins possibles défini donc le comportement d'un programme. On appelle *trace* cet ensemble. On notera que si le chemin σ est dans la trace d'un programme, $\mathrm{pref}(\sigma)$ est inclus dans cette trace : en effet, si un programme passe par toutes les étapes de σ, il passe aussi par les étapes de σ', si σ' est un préfixe de σ.

Une trace θ peut être infinie : dans le cas où le programme ne termine pas, il existe une suite de chemins $\sigma_0, \sigma_1, \cdots$ telle que $\forall i \in \mathbb{N}, \exists s_i \in \mathcal{S}, \sigma_{i+1} = \sigma_i.s_i$. Et tous ces chemins sont dans θ.

Définition 3.3.2 (Trace)**.** Une *trace* θ est un élément de $\mathcal{T} \stackrel{\text{def}}{=} \wp(\mathcal{S}^*)$ tel que θ est préfixe (Définition 1.1.2).

À chaque zone exécutable $\mathbf{x}$, on peut associer sa trace de la manière suivante.

Définition 3.3.3 (Trace d'une zone)**.** Si $\mathbf{x} \in \mathcal{Z}$ est une zone, alors sa trace $\Theta(\mathbf{x})$ est définie par

$$\Theta(\mathbf{x}) \stackrel{\text{def}}{=} \{s_0.\cdots.s_n \mid \exists \mathbf{d}, s_0 = \langle \mathbf{x}, 0, \mathbf{d}\rangle \wedge \forall i \in [\![1, n]\!], s_{i-1} \to s_i\}.$$

Si $\mathbf{d} \in \mathcal{Z}$ est une zone, on notera $\Theta(\mathbf{x}, \mathbf{d})$ la trace de $\mathbf{d}$ limitée à la donnée $\mathbf{d}$:

$$\Theta(\mathbf{x}, \mathbf{d}) \stackrel{\text{def}}{=} \{\langle \mathbf{x}, 0, \mathbf{d}\rangle.\sigma \in \Theta(\mathbf{x})\}$$

À partir d'une trace on va étudier les mouvements des registres pendant l'exécution

3.4 Cinématique des registres

3.4.1 Positions des registres

Définition 3.4.1 (Position)**.** Une *position* r est le couple $r = (z, n) \in \{X, D\} \times \mathbb{N}$. La position (X, n) désigne le n-ème registre de $\mathbf{x}$ et (D, n) le n-ème registre de $\mathbf{d}$. Par exemple, la position $(X, 0)$ désigne le registre étant exécuté.

Étant donné que le registre en tête de la zone exécutable $\mathbf{x}$ n'est pas désactivable, on lui attribue la position $(X, 0)$. La position de la tête de $\mathbf{d}$ est notés $(D, 1)$. Cette notation permet de commencer la numérotation des registres désactivables de $\mathbf{x}$ et des registres activables de $\mathbf{d}$ par l'indice 1.

On appelle $(X, 0)$ la *position d'exécution* et $(D, 1)$ la *position d'édition*.

3.4.2 Description des mouvements

On définit alors le *résiduel* r' d'une position r par un état s comme une position vérifiant $r\ [s]\ r'$. La relation $[s]$ est donnée dans la Figure 3.4. Par exemple, si s est un état de la forme

$$\langle p_n, \cdots, p_1 \mid r_0 \bullet \boxed{\overline{\texttt{act}} \bullet r'_0} \mid d_1, \cdots, d_m\rangle$$

(c'est à dire la prochaine instruction à exécuter est `act`), alors la position $(D, 1)$ va passer en position $(X, 1)$ par la relation $[s]$, car la donnée d_1 va passer juste devant p_1. De plus, comme d_1 est partie de $\mathbf{d}$, les autres positions dans $\mathbf{d}$ sont décrémentées (comme pour un `pop`). Donc les positions $(D, i+1)$ passent en position (D, i) par $[s]$. Enfin, comme on a ajouté d_1 juste avant p_1, tous les programmes p_i de $\mathbf{x}$ sont décalés d'un rang vers le bas. Donc les positions (X, i) passent en position $(X, i+1)$ par $[s]$.

On définit également une relation $\lceil s \rceil$ telle que

$$\lceil s \rceil \stackrel{\text{def}}{=} \{((m, n), (m, n')) \mid m \in \{X, D\}\ n, n' \in \mathbb{N}\} \cap [s]$$

L'idée est que $r\ \lceil s \rceil\ r'$ lorsque $r\ [s]\ r'$ et que r ne subit pas de comportement réflexif (il ne change pas de zone).

(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{act}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(X,i+1)$	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{act}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,i-1)$	$2 \leqslant i \leqslant m$
$(D,1)$	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{act}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(X,1)$	
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{inact}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,i+1)$	$1 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{inact}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(X,i-1)$	$2 \leqslant i \leqslant n$
$(X,1)$	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{inact}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,1)$	
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{right}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{right}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,i+1)$	$1 \leqslant i \leqslant m-1$
(D,m)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{right}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,1)$	
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{left}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{left}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,i-1)$	$2 \leqslant i \leqslant m$
$(D,1)$	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{left}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(D,m)	
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{pop}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{pop}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,i-1)$	$2 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{new}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{new}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	$(D,i+1)$	$1 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{put}\,\#} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{put}\,\#} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(D,i)	$1 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{put}\,1} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{put}\,1} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(D,i)	$1 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{jump}\,{+}j} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{jump}\,{+}j} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(D,i)	$1 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{jump}\,{-}j} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{jump}\,{-}j} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(D,i)	$1 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{case}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(X,i)	$1 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \bullet \boxed{\overline{\texttt{case}} \bullet r'_0} \mid r'_1,\cdots,r'_m\rangle]$	(D,i)	$1 \leqslant i \leqslant m$
(X,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \mid r'_1,\cdots,r'_m\rangle]$	$(X,i-1)$	$2 \leqslant i \leqslant n$
(D,i)	$[\langle r_n,\cdots,r_1 \mid r_0 \mid r'_1,\cdots,r'_m\rangle]$	(D,i)	$1 \leqslant i \leqslant m$

FIGURE 3.4 – Résiduel d'une position par un état

3.4.3 Déplacements pendant l'exécution

On étend les relations $[s]$ et $\lceil s \rceil$ sur les chemins : si σ est le chemin $[s_1, \cdots, s_n]$ on a $r_0\ [\sigma]\ r_n$ (resp. $r_0\ \lceil\sigma\rceil\ r_n$) quand il existe des positions $r_1, \cdots, r_{n-1}$ telles que, pour tout $i \in [\![1, n]\!]$, $r_{i-1}\ [s_i]\ r_i$ (resp. $r_{i-1}\ \lceil s_i \rceil\ r_i$). On a également, pour tout r, $r\ [\epsilon]\ r$ et $r\ \lceil\epsilon\rceil\ r$.

Finalement, on définit l'*image* et l'*image non réflexive* d'une liste d'états par

$$\begin{aligned} \Im([\sigma]) &\stackrel{\text{def}}{=} \{r' \mid \exists r, r\ [\sigma]\ r'\} \\ \Im(\lceil\sigma\rceil) &\stackrel{\text{def}}{=} \{r' \mid \exists r, r\ \lceil\sigma\rceil\ r'\} \end{aligned}$$

Exemple 3.4.2 (Exemple d'application). Si $(X, 0) \in \Im(\lceil\sigma\rceil)$ et que $\sigma = [s_0, \cdots, s_n]$ et que $s_0 = \langle \mathbf{x}_0, a_0, \mathbf{d}_0 \rangle$ et $s_n = \langle \mathbf{x}_n, a_n, \mathbf{d}_n \rangle$, alors, le registre en tête de $\mathbf{x}_n$ (registre exécutable) "provient" de $\mathbf{x}_0$, dans le sens où ce registre n'a fait que des mouvements dans la zone exécutable (il n'est pas passé dans la zone des données $\mathbf{d}$) pendant l'exécution σ. ■

3.4.4 Comportements réflexifs

Définition 3.4.3 (Chemin sans réflexion)**.** Un chemin σ est un *chemin sans réflexion*, noté $\mathsf{norefl}(\sigma)$, lorsque

$$\forall \sigma' \in \mathrm{pref}(\sigma) \quad (X, 0) \in \Im(\lceil\sigma'\rceil)$$

Dit autrement, σ est sans réflexion lorsque tous les registres en position d'exécution dans σ sont déjà dans la zone $\mathbf{x}$ avant σ. Sachant que seul le registre en position $(X, 0)$ peut être exécuté, un chemin est sans réflexion lorsqu'on exécute aucun registre provenant des données.

Définition 3.4.4 (Chemin sans réification)**.** Un chemin σ est un *chemin sans réification*, noté $\mathsf{noreif}(\sigma)$, lorsque

$$\forall \sigma' \in \mathrm{pref}(\sigma) \quad (D, 1) \in \Im(\lceil\sigma'\rceil)$$

Dit autrement, σ ne comporte pas de réification lorsque tous les registres en position d'édition dans σ sont déjà dans la zone $\mathbf{d}$ avant σ. Sachant que seul le registre en position $(D, 1)$ peut être lu et édité, un chemin ne comporte pas de réification lorsqu'on ne lit ou ne modifie aucun registre provenant des programmes.

Définition 3.4.5 (Programme sans réflexion)**.** Une zone $\mathbf{x}$ est *sans réflexion* lorsque

$$\forall \sigma \in \Theta(\mathbf{x}) \quad \mathsf{norefl}(\sigma)$$

c'est à dire lorsque tous les chemins de ses traces sont sans réflexion.

Définition 3.4.6 (Programme sans réification)**.** Une zone $\mathbf{x}$ est *sans réification* lorsque

$$\forall \sigma \in \Theta(\mathbf{x}) \quad \mathsf{noreif}(\sigma)$$

c'est à dire lorsque tous les chemins de ses traces sont sans réification.

La propriété suivante est trivialement vraie.

Propriété 3.4.7. *Si l'instruction* `act` *(resp.* `inact`*) n'est exécutée par aucun état d'un chemin de la trace de* $\mathbf{x}$*, alors* $\mathbf{x}$ *est sans réflexion (resp. sans réification).*

Par définition, un programme est avec réflexion lorsqu'il existe un chemin dans sa trace qui *exécute un registre provenant des données*, et il est avec réification lorsqu'il existe un chemin dans sa trace qui *lit ou modifie un registre provenant des programmes.*

3.5 Conclusion

On a détaillé un modèle de machine présentant de manière explicite les phénomènes réflexifs : la différence entre l'état de donnée et celle de programme est structurellement marquée dans la machine. Ainsi les changements d'états (qui correspondent l'un à une réflexion, l'autre à une réification) sont eux aussi structurels.

Profitant de cette approche "visuelle" de la réflexion, on a défini une mesure des mouvements réflexifs effectués dans cette machine, et à partir d'elle, on a pu définir ce qu'étaient une réflexion et une réification dans un programme. Dans le chapitre suivant, on va montrer une méthode extrayant les informations des réflexions d'un programme.

Chapitre 4

Abstraction sur la réflexion

On souhaite abstraire notre modèle de machine afin de ne garder que les informations des réflexions. Dans le chapitre précédent on a donné une définition des programmes sans réflexion (Définition 3.4.5) : il s'agit des programmes n'exécutant pas de registre provenant des données. Dans ce chapitre, on va justement s'intéresser aux programmes *avec* réflexions. Elles auront lieu lorsqu'un registre *initialement* dans les données sera exécuté.

Afin de "mesurer" le degré de réflexion d'un programme, on va décomposer sa trace en mettant en valeur chaque réflexion. C'est à dire que l'on va oublier toutes les informations non reliées à une réflexion. Concrètement, on va ne retenir que les moments de l'exécution où s'opèrent des réflexions. Cette abstraction est issue de la thèse de Reynaud [Rey10]. Elle a trouvé des applications pratiques dans l'étude de programmes auto-modifiants comme les virus informatiques [Bon+13 ; Bon+12].

La première section donne le principe de l'abstraction, et l'idée permettant d'abstraire une trace. Dans la seconde section on verra comment abstraire la sémantique concrète des traces pour ne garder que les informations des réflexions. Cette abstraction sera effectuée simplement en observant la structure de la machine, *sans avoir besoin de connaire la sémantique de la machine*. Puis on construira dans la dernière section des abstractions des règles de calcul de la machine, complètes pour la sémantique abstraite.

4.1 Principe

4.1.1 Stratégie

Dans cette section on donne un aperçu des notions que nous allons définir à partir de la section suivante. D'un coté on met au point une abstraction α faisant ressortir les réflexions des traces d'exécution. L'abstraction α est "*réduction agnostique*", c'est à dire qu'elle n'utilise pas les règles de réduction de la machine $s \to s'$ (Figure 3.3) pour abstraire une trace. D'autre part, on définit une sémantique abstraite $\xrightarrow{\#}$ nous permettant de définir une trace abstraite $\Theta^{\#}(\mathbf{x})$ à partir d'une zone exécutable $\mathbf{x}$. La correction de l'abstraction est vérifiée si

$$\alpha(\Theta(\mathbf{x})) \subseteq \Theta^{\#}(\mathbf{x})$$

4.1.2 Idées

4.1.2.1 Abstraction de trace

Un chemin $\sigma = s_0, \cdots, s_n$ est une séquence d'états. Il décrit l'ensemble des informations portées par le calcul. Dans ce qui suit, on ne souhaite garder que les informations propres aux réflexions dans le calcul. Un chemin est découpé en m chemins $\sigma_1, \cdots, \sigma_m$ (Équation (4.1)) tels que, pour tout i, σ_i soit avec réflexions (Équation (4.3)), mais que tous ses sous-chemins stricts ne le soient pas (Équation (4.2)).

$$\sigma = \sigma_1 \bullet \ldots \bullet \sigma_m \tag{4.1}$$

$$\forall i \in [\![1, m]\!] \quad \sigma_i = \sigma_i'.s \Rightarrow \mathsf{norefl}(\sigma_i') \tag{4.2}$$

$$\forall i \in [\![1, m]\!] \quad \neg\mathsf{norefl}(\sigma_i) \tag{4.3}$$

Pour chaque segment σ_i on retient le programme responsable de ce chemin. C'est $\mathbf{x}_i$, où $\sigma_i = \langle \mathbf{x}_i, 0, \mathbf{d}_i \rangle.\sigma_i'$. On appelle $\mathbf{x}_i$ le *témoin* du chemin σ_i. L'abstraction $\alpha(\sigma)$ est définie par

$$\alpha(\sigma) \stackrel{\text{def}}{=} [\mathbf{x}_1, \cdots, \mathbf{x}_m]$$

Dans une trace abstraite, on cherche à isoler tous les différents codes qui ont été exécutés pendant le calcul. Comme on étudie les programmes réflexifs, l'ensemble du code exécutable n'est pas connu avant l'exécution. La trace abstraite nous donne la *cinématique* d'apparition des codes exécutables. L'idée est que, dans le cas d'un programme non réflexif dont le code est $\mathbf{x}$, sa trace abstraite se limite au singleton $\{[\mathbf{x}]\}$.

4.1.2.2 Sémantique abstraite

Chaque témoin $\mathbf{x}_i$ d'un chemin σ_i produit en s'exécutant le chemin σ_i, c'est à dire encore

$$\sigma_i \in \Theta(\mathbf{x}_i)$$

À partir de $\mathbf{x}_i$, en utilisant $\Theta(\mathbf{x}_i)$, on peut alors calculer $\mathbf{x}_{i+1}$. En effet $\sigma_i \bullet \sigma_{i+1} \in \Theta(\mathbf{x}_i)$. On sait que σ_i est le plus petit chemin avec réflexions du chemin $\sigma_i \bullet \sigma_{i+1}$. On connait donc "où commence" σ_{i+1}, et on en déduit $\mathbf{x}_{i+1}$. On notera $\mathbf{x}_i \xrightarrow{\#} \mathbf{x}_{i+1}$. On a alors

$$\alpha(\sigma) \subseteq \Theta^{\#}(\mathbf{x}_1)$$

où

$$\Theta^{\#}(\mathbf{x}) \stackrel{\text{def}}{=} \{[\mathbf{x}_0, \cdots, \mathbf{x}_n] \mid \mathbf{x}_0 = \mathbf{x} \wedge \forall i \in [\![1, n]\!], \mathbf{x}_{i-1} \xrightarrow{\#} \mathbf{x}_i\}$$

La suite de ce chapitre va définir chacune de ses notions (α et $\xrightarrow{\#}$) et montrer la correction de α par rapport à $\xrightarrow{\#}$.

4.1.3 Tour de réflexions

Dans la Section 2.4, on notait $p_1 \rightsquigarrow p_2$ lorsque le programme p_1 lançait l'exécution du programme p_2, sous forme d'une tour de réflexions. L'idée était de dire que p_1 s'exécutait sans réflexion, puis passait la main à p_2.

$$\textsc{AbstrInit}\ \frac{s = \langle \mathbf{x}, 0, \mathbf{d} \rangle}{[s] \vdash [s] : [\mathbf{x}]} \qquad \textsc{AbstrNoRefl}\ \frac{\sigma' \vdash \sigma : \Sigma \qquad \mathsf{norefl}(\sigma')}{\sigma'.s \vdash \sigma.s : \Sigma}$$

$$\textsc{AbstrRefl}\ \frac{\sigma' \vdash \sigma : \Sigma \qquad \neg\mathsf{norefl}(\sigma') \qquad s = \langle \mathbf{x}, 0, \mathbf{d} \rangle}{[s] \vdash \sigma.s : \Sigma.\mathbf{x}}$$

FIGURE 4.1 – Abstraction de trace

Dans ce chapitre, la règle $\rightsquigarrow$ peut être comparée à la règle $\xrightarrow{\#}$, dans le cas où on ne s'intéresse pas à la réification. En effet, $\mathbf{x}_1 \xrightarrow{\#} \mathbf{x}_2$ dénote qu'une exécution sans réflexion de $\mathbf{x}_1$ conduit à l'exécution de $\mathbf{x}_2$, qui n'est pas issu de $\mathbf{x}_1$. Et $p_1 \rightsquigarrow p_2$ exprime le fait que p_1 lance l'exécution de p_2.

4.2 Abstraction

4.2.1 Chemin abstrait

L'abstraction d'un chemin, c'est à dire une liste d'états, est une liste de zones.

Définition 4.2.1 (Chemin abstrait)**.** Un *chemin abstrait* est un élément de $\mathcal{Z}^*$, où on rappelle que $\mathcal{Z}$ est l'ensemble des listes de registres.

4.2.2 Trace abstraite

Définition 4.2.2 (trace abtraite)**.** Une *trace abstraite* $\theta^\#$ est un élément de $\mathcal{T}^\# \stackrel{\text{def}}{=} \wp(\mathcal{Z}^*)$ tel que $\theta^\#$ est préfixe (Définition 1.1.2).

4.2.3 Abstraction de trace

4.2.3.1 Définition de l'abstraction

On va construire une abstraction $\alpha : \mathcal{S}^* \to \mathcal{Z}^*$ associant à σ son abstraction $\alpha(\sigma)$. Si θ est une trace, alors sa trace abstraite associée sera

$$\alpha(\theta) \stackrel{\text{def}}{=} \{\alpha(\sigma) \mid \sigma \in \theta\}$$

Dans la Section 4.1, on a donné les propriété de l'abstraction α. On donne ici une manière algorithmique de construire un α répondant à ces propriétés. On utilise les règles d'inférences de la Figure 4.1. Les règles sont de la forme $\sigma' \vdash \sigma : \Sigma$, où $\sigma, \sigma' \in \mathcal{S}^*$ et $\Sigma \in \mathcal{Z}^*$. La liste Σ est l'abstraction de σ, et on pose

$$\alpha(\sigma) \stackrel{\text{def}}{=} \Sigma$$

La règle AbstrInit initialise l'abstraction à la première zone exécutable de σ. La règle AbstrNoRefl vérifie que le registre en position d'être exécuté (position $(X, 0)$) n'a jamais fait partie des données (il est tout le temps resté dans la zone exécutable $\mathbf{x}$). Si c'est effectivement le cas, alors l'abstraction de $\sigma.s$ est la même que celle de σ. Dans le cas contraire (règle AbstrRefl) la zone exécutable courante $\mathbf{x}$ est ajoutée à l'abstraction Σ de σ.

Le chemin σ' correspond au début d'un des σ_i de la Section 4.1.

Propriété 4.2.3. *Si $\sigma' \vdash \sigma : \Sigma$ est dérivable, il existe σ'' tel que $\sigma = \sigma'' \bullet \sigma'$.*

Démonstration. Par induction sur la dérivation. □

Tout sous-chemin strict de σ' est sans réflexion.

Propriété 4.2.4. *Si $\sigma'.s \vdash \sigma.s : \Sigma$ est dérivable, alors* $\mathsf{norefl}(\sigma')$.

Démonstration. Par induction sur la dérivation. □

Propriété 4.2.5. *Si $\sigma = \langle \mathbf{x}, a, \mathbf{d} \rangle.\sigma'$ et* $\mathsf{norefl}(\sigma)$, *alors $\alpha(\sigma) = [\mathbf{x}]$*

Démonstration. On montre par induction que $\forall \sigma' \in \mathrm{pref}(\sigma), \sigma' \vdash \sigma' : [\mathbf{x}]$. □

4.2.3.2 Exemple d'abstraction

Supposons que l'on dispose des codes suivants ($\mathbf{x} \stackrel{\text{def}}{=} [r_1, r_2, r_3]$ et $\mathbf{d} \stackrel{\text{def}}{=} [r_4, r_5, r_6]$) :

$\mathbf{x}$	$\mathbf{d}$
$r_1 \stackrel{\text{def}}{=} \overline{\texttt{act}} \bullet \overline{\texttt{act}}$	$r_4 \stackrel{\text{def}}{=} \overline{\texttt{act}}$
r_2	$r_5 \stackrel{\text{def}}{=} \epsilon$
r_3	r_6

Lors de l'exécution, le code r_1 commence et active les codes r_4 et r_5.

$$\underbrace{\langle r_3, r_2 \mid \boxed{\overline{\texttt{act}} \bullet \overline{\texttt{act}}} \mid r_4, r_5, r_6 \rangle}_{s_0} \to \underbrace{\langle r_3, r_2, r_4 \mid \overline{\texttt{act}} \bullet \boxed{\overline{\texttt{act}}} \mid r_5, r_6 \rangle}_{s_1} \to \underbrace{\langle r_3, r_2, r_4, r_5 \mid \overline{\texttt{act}} \bullet \overline{\texttt{act}} \mid r_6 \rangle}_{s_2}$$

Puis on *change de registre d'exécution* : le dernier code a avoir été activé est exécuté.

$$\underbrace{\langle r_3, r_2, r_4, r_5 \mid \overline{\texttt{act}} \bullet \overline{\texttt{act}} \mid r_6 \rangle}_{s_2} \to \underbrace{\langle r_3, r_2, r_4 \mid \boxed{r_5} \mid r_6 \rangle}_{s_3}$$

Il lance l'exécution du code suivant sur la pile $\mathbf{x}$.

$$\underbrace{\langle r_3, r_2, r_4 \mid \boxed{r_5} \mid r_6 \rangle}_{s_3} \to \underbrace{\langle r_3, r_2 \mid \boxed{r_4} \mid r_6 \rangle}_{s_4}$$

Ce dernier active le dernier code r_6 et lance son exécution.

$$\underbrace{\langle r_3, r_2 \mid \boxed{r_4} \mid r_6 \rangle}_{s_4} \to \underbrace{\langle r_3, r_2, r_6 \mid r_4 \mid \rangle}_{s_5} \to \underbrace{\langle r_3, r_2 \mid \boxed{r_6} \mid \rangle}_{s_6}$$

Posons $\sigma = [s_0, \cdots, s_6]$ et calculons $\alpha(\sigma)$. Posons $\sigma_i = [s_0, \cdots, s_i]$ pour $i \in [\![0, 6]\!]$.

1. Le premier élément de $\alpha(\sigma)$ est la première zone exécutable de σ, c'est à dire $\mathbf{x}_0 \stackrel{\text{def}}{=} [r_1, r_2, r_3]$. Donc
$$\sigma_0 \vdash \sigma_0 : [\mathbf{x}_0]$$
(règle AbstrInit).
2. L'état s_0 ne change pas le registre exécutable (r_1). Donc $(X, 0) \in \Im(\lceil \sigma_0 \rceil)$. De même, $(X, 0) \in \Im(\lceil \sigma_1 \rceil)$ car s_1 ne change pas non plus de registre exécutable. Donc
$$\sigma_2 \vdash \sigma_2 : [\mathbf{x}_0]$$
(règle AbstrNoRefl).
3. L'état s_3 a pour registre exécutable r_5 qui n'est pas issu de $\mathbf{x}_0$. Donc, si $\mathbf{x}_1 \stackrel{\text{def}}{=} [r_5, r_4, r_2, r_3]$
$$[s_3] \vdash \sigma_3 : [\mathbf{x}_0, \mathbf{x}_1]$$
(règle AbstrRefl).
4. Les états s_4 et s_5 ont pour registre exécutable r_4 qui provient de $\mathbf{x}_1$. Donc c'est la règle AbstrNoRefl qui s'applique.
$$[s_5, s_4, s_3] \vdash \sigma_5 : [\mathbf{x}_0, \mathbf{x}_1]$$
5. L'état s_6 exécute le registre r_6 qui n'est pas dans $\mathbf{x}_1$. C'est donc la règle AbstrRefl qui s'applique. Si $\mathbf{x}_2 \stackrel{\text{def}}{=} [r_2, r_3]$, alors
$$[s_6] \vdash \sigma_6 : [\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2]$$

On a donc
$$\alpha(\sigma) = [\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2]$$

4.3 Sémantique abstraite

4.3.1 Transition abstraite

On définit à présent une sémantique abstraite $\stackrel{\#}{\rightarrow}$ ne faisant ressortir que les réflexions. Elle est définie de la manière suivante : soit $s' = \langle \mathbf{x}', a', \mathbf{d}' \rangle$

$$\mathbf{x} \stackrel{\#}{\rightarrow} \mathbf{x}' \stackrel{\text{def}}{\iff} \begin{cases} \sigma.s.s' \in \Theta(\mathbf{x}) & (1) \\ \mathsf{norefl}(\sigma) & (2) \\ \neg\mathsf{norefl}(\sigma.s) & (3) \end{cases}$$

L'idée est la suivante

1. La zone exécutable $\mathbf{x}'$ est atteignable depuis $\mathbf{x}$.
2. Toutes les zones atteignables avant $\mathbf{x}'$ à partir de $\mathbf{x}$ exécutent des registres qui proviennent de $\mathbf{x}$.
3. La zone $\mathbf{x}'$ exécute un registre qui ne provient pas de $\mathbf{x}$.

4.3.2 Correction de la sémantique abstraite

On va maintenant montrer que cette relation est une bonne exécution abstraite de $\rightarrow$ par rapport à l'abstraction α.

Théorème 4.3.1. *Pour toute zone* $\mathbf{x}$

$$\alpha(\Theta(\mathbf{x})) \subseteq \Theta^{\#}(\mathbf{x})$$

Démonstration. On va montrer que pour tout $\sigma \in \Theta(\mathbf{x}_0)$, $\alpha(\sigma) \in \Theta^{\#}(\mathbf{x}_0)$. Par induction sur $\sigma'.s \vdash \sigma.s : \Sigma.\mathbf{x}$ on montre le résultat plus général

$$\alpha(\sigma \bullet s) \in \Theta^{\#}(\mathbf{x}_0) \tag{4.4}$$
$$\mathsf{noreif}(\sigma') \tag{4.5}$$
$$\sigma'.s \in \Theta(\mathbf{x}) \tag{4.6}$$

— Si la dernière règle utilisée est la règle AbstrInit, alors $\Sigma \in \Theta^{\#}(\mathbf{x}_0)$ car si $[s] \in \Theta(\mathbf{x}_0)$ alors il existe $\mathbf{d}$ tel que $s = \langle \mathbf{x}_0, 0, \mathbf{d} \rangle$. Donc $\Sigma = [\mathbf{x}_0] \in \mathcal{T}^{\#}(\mathbf{x}_0)$. De plus, $\sigma' = \epsilon$, donc $\forall \sigma'' \in \mathrm{pref}(\sigma'), (X, 0) \in \Im(\lceil \sigma'' \rceil)$. Enfin, comme $\mathbf{x} = \mathbf{x}_0$, $[s] \in \Theta(\mathbf{x})$.

— Si la dernière règle est la règle AbstrNoRefl, alors on a directement que $\Sigma.\mathbf{x} \in \Theta^{\#}(\mathbf{x}_0)$. De plus, on sait que si $\sigma' = \sigma''.s'$, alors $\forall \sigma^{(3)} \in \mathrm{pref}(\sigma''), (X, 0) \in \Im(\lceil \sigma^{(3)} \rceil)$. De plus on sait que $(X, 0) \in \Im(\lceil \sigma' \rceil)$. Donc $\forall \sigma^{(4)} \in \mathrm{pref}(\sigma'), (X, 0) \in \Im(\lceil \sigma^{(4)} \rceil)$. Enfin, comme $\sigma' \in \Theta(\mathbf{x})$, et $\sigma.s \in \Theta(\mathbf{x}_0)$ et $\sigma = \sigma \bullet u$, alors $\sigma'.s \in \Theta(\mathbf{x})$.

— Si la dernière règle est AbstrRefl, alors on sait que $\Sigma \in \Theta^{\#}(\mathbf{x}_0)$. Posons $\Sigma = \Sigma'.\mathbf{x}'$. De plus, on sait que si $\sigma' = \sigma''.s'$, alors $\forall \sigma^{(3)} \in \mathrm{pref}(\sigma''), (X, 0) \in \Im(\lceil \sigma^{(3)} \rceil)$. On sait aussi que $\sigma''.s' \in \Theta(\mathbf{x}')$, et donc $\sigma''.s'.s \in \Theta(\mathbf{x}')$. Enfin $(X, 0) \notin \Im(\lceil \sigma''.s' \rceil)$. Or $s = \langle \mathbf{x}, 0, \mathbf{d} \rangle$. Donc $\mathbf{x}' \xrightarrow{\#} \mathbf{x}$. Donc $\Sigma.\mathbf{x} \in \Theta^{\#}(\mathbf{x}_0)$. Les deux autres conditions sont facilement vérifiées.

□

4.4 Conclusion

On a donné une vue abstraite d'une exécution en partant des traces (concrètes) d'une zone exécutable $\mathbf{x}$ en ne retenant que les informations des réflexions : si σ est un chemin dans la trace, on découpe σ en sections σ' sans réflexion, c'est à dire que le registre exécuté de chaque étape de σ' provient de la zone exécutable initiale $\mathbf{x}'$ de $\sigma' = \langle \mathbf{x}', 0, \mathbf{d}' \rangle$.

De plus, l'abstraction des traces est correcte : chaque chemin σ' est représenté par son témoin $\mathbf{x}'$ et σ' est issu de $\mathbf{x}'$, c'est à dire que $\sigma' \in \Theta(\mathbf{x}')$.

Les témoins $\mathbf{x}'$ récapitulent l'ensemble des codes exécutés par la machine, et chacun correspond à une partie sans réflexion de l'exécution complète.

La succession entre les témoins $\mathbf{x}_1 \xrightarrow{\#} \mathbf{x}_2$ est comparable celle des programmes dans une tour de réflexions $p_1 \rightsquigarrow p_2$, lorsque l'on considère que p_1 et p_2 sont des programmes non réflexifs.

L'idée est la même que dans [Rey10] où la trace d'un programme est décomposée en sections sans réflexions appelées *vagues*.

Deuxième partie

Interprétation logique

Chapitre 5

Introduction : λ-calcul

On introduit notre cadre de travail pour la suite. On va s'intéresser au rapport entre calcul et logique. On va donner une interprétation logique de comportements réflexifs. Notre cadre de travail est donc centré autour de la correspondance de Curry-Howard : d'une part on décrit un langage ayant les capacités calculatoires que l'on souhaite, de l'autre on associe un système logique où les preuves sont encodées dans les termes de notre langage.

Ce chapitre a donc pour vocation de donner le minimum vital pour comprendre cette correspondance. Il présente le λ-calcul, ses principales propriétés (localité des variables, confluence, ...), puis montre comment il peut interpréter un système logique.

Ce chapitre sera aussi notre "modèle de rédaction". Il est un aperçu en miniature de la méthode des prochains chapitres dans lesquels on définira un langage réflexif exprimant un système logique. Il réintroduit les notions bien établies (renommage, substitution, types, ...) que l'on va utiliser dans cette thèse. Il est aussi l'occasion de donner les notations pour la suite.

Dans les chapitres suivants (notamment dans les Sections 9.5 et 11.4.3) on montrera que le système logique de ce calcul (la Logique Intuitionniste) peut être encodé dans les langages Λ_R et Λ_R^K décrits dans les Chapitres 7 et 11.

5.1 Lambda-calcul pur

Le λ-calcul est un langage de programmation inventé par Alonso Church dans les années 1930.

5.1.1 Syntaxe

Soit $\mathcal{V}$ un ensemble de variables. La syntaxe du langage est définie par

$$\Lambda \ni t \stackrel{\text{def}}{=} x \mid t\ t \mid \lambda x.t$$

où $x \in \mathcal{V}$.

5.1.2 Contextes de structure

On définit aussi les contextes de structure S qui permettent d'exprimer la position d'un sous-terme dans un autre.

$$S \stackrel{\text{def}}{=} [] \mid S\ t \mid t\ S \mid \lambda x.S$$

Si t est un terme, on note $S[t]$ le terme construit à partir de S en remplaçant [] par t. On note également $S_1 \circ S_2$ la composition de deux contextes de structure :

$$\begin{array}{rcl} [] \circ S' & \stackrel{\text{def}}{=} & S' \\ (S\ t) \circ S' & \stackrel{\text{def}}{=} & (S \circ S')\ t \\ (t\ S) \circ S' & \stackrel{\text{def}}{=} & t\ (S \circ S') \\ (\lambda x.S) \circ S' & \stackrel{\text{def}}{=} & \lambda x.(S \circ S') \end{array}$$

5.1.3 Liaison des variables

On note $\mathrm{FV}(t)$ l'ensemble des variables libres de t. Formellement, $\mathrm{FV}(t)$ se définit de la manière suivante :

$$\begin{array}{rcl} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(t_1\ t_2) & = & \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\ \mathrm{FV}(\lambda x.t) & = & \mathrm{FV}(t) \setminus \{x\} \end{array}$$

Par exemple y est libre dans $\lambda x.x\ y$ mais x ne l'est pas.

5.1.4 α-renommage

On considère à présent les termes modulo α-renommage, c'est à dire qu'on peut substituer à loisir les variables liées des termes.

Exemple 5.1.1. Le terme $\lambda x.\lambda x.x$ est α-équivalent à $\lambda x.\lambda y.y$. ■

5.1.5 Substitution

La relation de réécriture du λ-calcul utilise la substitution des variables libres d'un terme. On note $t[x/t']$ la substitution de x par t' dans t. On la définit de la manière suivante

$$\begin{array}{rcll} x[x/t'] & \stackrel{\text{def}}{=} & t' & \\ y[x/t'] & \stackrel{\text{def}}{=} & y & x \neq y \\ (\lambda x.t)[x/t'] & \stackrel{\text{def}}{=} & \lambda x.t & \\ (\lambda y.t)[x/t'] & \stackrel{\text{def}}{=} & \lambda y.t[x/t'] & x \neq y \wedge y \notin \mathrm{FV}(t') \\ (t_1\ t_2)[x/t'] & \stackrel{\text{def}}{=} & t_1[x/t']\ t_2[x/t'] & \end{array}$$

On note que la substitution ne s'effectue, par définition, que sur les variables libres des termes. La condition $y \notin \mathrm{FV}(t')$ évite le phénomène de *capture de variable* : par exemple, la substitution $(\lambda y.x)[x/y] = \lambda y.y$ (ne correspond pas à la définition précédente) n'est pas correcte car elle capture la variable libre x par le lieur λy. La condition peut toujours être vérifiée en réécrivant $\lambda y.t$ par α-renommage.

5.1.6 Contextes de réduction

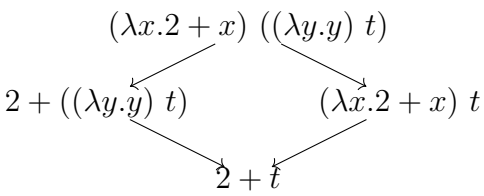
Les *contextes de réduction* décrivent les positions des sous-termes qu'il est autorisé de réduire. Dans le cas du λ-calcul, on peut réduire n'importe quel sous-terme. Donc, un contexte de réduction E est un contexte de structure.

5.1.7 β-réduction

La β-réduction est définie de la manière suivante : si $t_0 = E[(\lambda x.t)\ t']$ et $t_1 = E[t[x/t']]$, alors t_0 se réduit à t_1, ce qu'on note $t_0 \to t_1$. On note $\xrightarrow{*}$ la clôture réflexive et transitive de $\to$. Lorsqu'il n'existe pas de terme t_1 tel que $t_0 \to t_1$, on dit que t_0 est en *forme normale*. On note $=_\beta$ la plus petite relation d'équivalence issue de $\to$.

5.1.8 Confluence

Beaucoup de termes du système ont plusieurs chemins de réduction. Par exemple, le terme $(\lambda x.x)\ ((\lambda y.y)\ t)$ peut se réduire de (au moins) deux manières.



Malgré cette différence, les deux chemins finissent par converger sur $2 + t$. Le système de réécriture est *confluent*, c'est à dire que si $t \xrightarrow{*} t_1$ et $t \xrightarrow{*} t_2$, il existe t' tel que $t_1 \xrightarrow{*} t'$ et $t_2 \xrightarrow{*} t'$. L'intérêt de cette propriété est que la sémantique d'un terme ne dépend pas de la façon dont on le réduit.

5.1.9 Stratégies de réduction

Le fait qu'il soit possible de passer par plusieurs chemins de réécriture pour réduire un terme n'est pas une propriété souhaitée pour un langage de programmation. En effet, en particulier, un terme peut à la fois se réduire sur une forme normale, et se réduire infiniment. Par exemple, le terme $((\lambda x.\lambda y.x)\lambda x.x)((\lambda x.xx)\lambda x.xx)$ peut se réduire en deux coups :

$$\underline{((\lambda x.\lambda y.x)\lambda x.x)}((\lambda x.xx)\lambda x.xx) \to \underline{(\lambda y.\lambda x.x)((\lambda x.xx)\lambda x.xx)} \to \lambda x.x$$

alors qu'il se réduit infiniment si on commence par $(\lambda x.xx)\lambda x.xx$.

On donne alors au langage une *stratégie de réduction*, c'est à dire on définit une relation de réduction $\to'$ telle que $\to' \subseteq \to$ et si $t \to' t_1$ et $t \to' t_2$, alors $t_1 = t_2$.

On définit une stratégie en choisissant au plus un redex dans chaque terme du langage. Pour cela, on peut utiliser des *contextes d'exécution* Φ, qui, à la manière des contextes de réduction E sont des termes à trou, mais définis de telle manière que pour un terme t donné, il existe au plus un couple (Φ, t') tel que $t = \Phi[t']$ et t' est un redex.

Il existe plusieurs façons de définir une stratégie. Plotkin utilise la sémantique opérationnelle structurelle (ou sémantique à petits pas) dans [Plo81]. On rappellera pour mémoire la sémantique naturelle (ou sémantique à grands pas) de Kahn dans [Kah87].

Dans ce travail, on utilise la méthode de Felleisen et Wright dans [WF94] pour définir la stratégie de réduction en *appel par nom* à partir des contextes d'exécution suivant :

$$\Phi \coloneqq [] \mid \Phi\ t.$$

On notera $\overset{\text{CBN}}{\to}$ la relation de réécriture en stratégie d'appel par nom. On remarquera que cette fois ci, il n'est pas possible de réduire sous les λ. L'intérêt est que si l'on part de la réduction de termes clos, tous les redexes de la réduction sont eux même clos. On ne risque donc pas de devoir réduire des termes de la forme $x\ t$ qui bloqueraient la réduction. Par contre, une conséquence est que l'ensemble des formes normales pour $\to$ est strictement inclus dans celui des formes normales pour $\overset{\text{CBN}}{\to}$: $\lambda x.((\lambda y.y)\lambda z.z)$ est une forme normale pour $\overset{\text{CBN}}{\to}$ mais pas pour $\to$.

Pour mémoire, il est aussi possible de définir la stratégie de réduction en *appel par valeur*, à partir des contextes d'exécution suivants

$$\Phi \coloneqq [] \mid \Phi\ t \mid (\lambda x.t)\ \Phi$$

et en réduisant la contraction du redex aux cas de la forme

$$\Phi[(\lambda x.t)(\lambda y.t')] \overset{\text{CBV}}{\to} \Phi[t[x/\lambda y.t']]$$

5.2 Interprétation logique : lambda-calcul simplement typé

5.2.1 Logique Intuitionniste

Le λ-calcul peut encoder un système de preuves. C'est la correspondance de Curry-Howard. Ce système de preuves est la Logique Intuitionniste, ou plus précisément la Logique Implicative Minimale [Joh37]. Les formules sont données par

$$A, B \overset{\text{def}}{=} \alpha \mid A \to B$$

On la rappelle dans la Figure 5.1 en déduction naturelle. Les jugements sont de la forme $\Gamma \vdash_n A$, où Γ est un ensemble de formules. Ce système de preuves capture les formules de la Logique Intuitionniste (sans les opérateurs $\wedge$ et $\vee$). Ce système est équivalent à la version en calcul des séquents donnée dans la Figure 5.2 [GTL89].

Théorème 5.2.1. *Le jugement $\Gamma \vdash_n A$ est dérivable si et seulement si le jugement $\Gamma \vdash_s A$ l'est.*

$$\frac{}{\Gamma, A \vdash_n A} \qquad \frac{\Gamma, A \vdash_n B}{\Gamma \vdash_n A \to B}$$

$$\frac{\Gamma \vdash_n A \to B \qquad \Gamma \vdash_n A}{\Gamma \vdash_n B}$$

FIGURE 5.1 – Logique Implicative Minimale en déduction naturelle

$$\frac{}{A \vdash_s A} \qquad \frac{\Gamma \vdash_s A \qquad \Delta, A \vdash_s B}{\Gamma, \Delta \vdash_s B}$$

$$\frac{\Gamma \vdash_s A \qquad \Delta, B \vdash_s C}{\Gamma, \Delta, A \to B \vdash_s C} \qquad \frac{\Gamma, A \vdash_s B}{\Gamma \vdash_s A \to B}$$

$$\frac{\Gamma, A, A \vdash_s B}{\Gamma, A \vdash_s B} \qquad \frac{\Gamma \vdash_s B}{\Gamma, A \vdash_s B}$$

FIGURE 5.2 – Logique Implicative Minimale en calcul des séquents

5.2.2 Syntaxe pour la Logique Intuitionniste

Le λ-calcul fait un bon terme de preuve pour la Logique Intuitionniste. Chaque formule A devient un type d'un terme, et chaque formule de l'environnement Γ est associée à une variable libre de ce terme. On note $\Gamma = x_1 : A, \cdots, x_n : A_n$ cette association, où toutes les variables λx_i sont distinctes (on pourra s'en assurer par α-renommage). Un jugement est de la forme $\Gamma \vdash_n t : A$ (resp. $\Gamma \vdash_s t : A$), le système de types en déduction naturelle (resp. calcul des séquents) est donné par la Figure 5.3 (resp. 5.4). Ces deux systèmes sont encore équivalents

Théorème 5.2.2. *Le jugement $\Gamma \vdash_n t : A$ est dérivable si et seulement si le jugement $\Gamma \vdash_s t : A$ l'est.*

Par exemple, la dérivation

$$\frac{\Gamma \vdash_n t_1 : A \to B \qquad \Gamma \vdash_n t_2 : A}{\Gamma \vdash_n t_1\ t_2 : B}$$

devient

$$\frac{\dfrac{\Gamma \vdash_s t_1 : A \to B \qquad \dfrac{\Gamma \vdash_s t_2 : A \qquad x : A \vdash_s x : A}{\Gamma, f : A \to B \vdash_s f\ t_2 : B}}{\Gamma, \Gamma \vdash_s t_1\ t_2 : B}}{\Gamma \vdash_s t_1\ t_2 : B}$$

On a donc associé *à chaque preuve en Logique Intuitionniste* un terme en λ-calcul, dont la preuve est sa dérivation de typage. L'ensemble des termes bien typés (ceux ayant une

$$\frac{}{\Gamma, x : A \vdash_n x : A} \qquad \frac{\Gamma, x : A \vdash_n t : B}{\Gamma \vdash_n \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash_n t_1 : A \to B \qquad \Gamma \vdash_n t_2 : A}{\Gamma \vdash_n t_1\ t_2 : B}$$

FIGURE 5.3 – Typage du λ-calcul en déduction naturelle

$$\frac{}{x : A \vdash_s x : A} \qquad \frac{\Gamma \vdash_s t' : A \qquad \Delta, x : A \vdash_s t : B}{\Gamma, \Delta \vdash_s t[x/t'] : B}$$

$$\frac{\Gamma \vdash_s t' : A \qquad \Delta, x : B \vdash_s t : C}{\Gamma, \Delta, f : A \to B \vdash_s t[x/f\ t'] : C} \qquad \frac{\Gamma, x : A \vdash_s t : B}{\Gamma \vdash_s \lambda x.t : A \to B}$$

$$\frac{\Gamma, y : A, z : A \vdash_s t : B}{\Gamma, x : A \vdash_s t[y/x][z/x] : B} \qquad \frac{\Gamma \vdash_s t : B}{\Gamma, x : A \vdash_s t : B}$$

FIGURE 5.4 – Typage du λ-calcul en calcul des séquents

dérivation de typage) est un sous-ensemble strict de l'ensemble des termes du λ-calcul. Par exemple, le terme $\lambda x.xx$ n'est pas bien typé.

Cet ensemble est stable par réduction.

Théorème 5.2.3 (Réduction du sujet)**.** *Si* $t \to t'$ *et* $\Gamma \vdash t : A$*, alors* $\Gamma \vdash t' : A$*.*

5.3 Conclusion

Dans ce chapitre introductif, on a montré l'exemple original d'association d'un système logique avec un langage de programmation. À partir de cet exemple, on a donné les notations communes que nous utiliserons dans tous les langages de cette partie.

Chapitre 6

Logique linéaire

L'utilisation du cadre de la Logique Linéaire avec le langage Λ_R donne une nouvelle interprétation de la modalité !, qui ne sera plus utilsée pour le contrôle des ressources, mais pour la séparation des données et des programmes. D'autre part, certains phénomènes observables en Logique Linéaire trouveront echo dans notre travail dans un contexte tout autre. Le problème de la cohérence, par exemple, est lié aux problèmes de confluence dans un langage avec des programmes gelés (sous forme de données) (voir Section 7.3.2).

6.1 Introduction

6.1.1 Gestion des ressources

L'idée de la Logique Linéaire est de comptabiliser les ressources *consommées* dans la preuve. Par *consommé* on entend *utilisé dans la règle de l'axiome* :

$$\textsc{Axiom}\ \frac{}{A \vdash A}$$

Par exemple, le séquent $A, B \vdash A$ n'est *a priori* pas dérivable, car l'hypothèse B ne serait pas utilisée. C'est pourquoi, les règles de contraction et d'affaiblissement sont interdites dans le cas général.

D'autre part, le partage des ressources dans la dérivation d'une preuve est elle aussi contrôlée : en logique classique, le connecteur $\wedge$ peut donner lieu à deux règles de dérivation

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \quad \text{ou} \quad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Dans la première version (dite multiplicative), les ressources sont partagées entre les deux branches de la preuve, il n'y a donc pas de création de ressources, alors que dans la deuxième (dite additive), on garde les mêmes ressources dans les deux branches : chacune des ressources est donc dupliquée. Cette distinction donne lieu à deux connecteurs conjonctifs : $\otimes$ (multiplicatif) et & (additif). De même la disjonction $\vee$ est également découpée en deux connecteurs $\parr$ (multiplicatif) et $\oplus$ (additif).

$$\textsc{Axiom}\ \frac{}{A \vdash A} \qquad \textsc{Cut}\ \frac{\Gamma \vdash B \qquad \Delta, B \vdash A}{\Delta, \Gamma \vdash A} \qquad \textsc{Contraction}\ \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \textsc{Weakening}\ \frac{\Gamma \vdash B}{\Gamma, !A \vdash B}$$

$$\multimap\text{-}\textsc{Right}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \multimap\text{-}\textsc{Left}\ \frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} \qquad \textsc{Dereliction}\ \frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \qquad \textsc{Promotion}\ \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}$$

FIGURE 6.1 – Logique Linéaire Intuitionniste

6.1.2 Exponentielles

Afin d'encadrer les duplications structurelles des ressources (c'est à dire les duplications non liées à des connecteurs logiques), on utilise les *exponentielles* ! et ?. La première décrit une ressource que l'on peut utiliser à volonté (les duplications (contractions) sont autorisées) ou supprimer (affaiblissement), et la deuxième est son dual : $\neg !A = ?\neg A$ et $\neg ?A = !\neg A$.

6.1.3 Logique Linéaire Intuitionniste

Nous nous limiterons dans ce travail à l'étude de la logique linéaire intuitionniste (ILL). D'autre part nous n'étudierons pas le comportement calculatoire des opérateurs $\otimes, \oplus, \&$ et $⅋$. Nous ne nous intéresserons qu'à l'exponentielle ! de la logique linéaire, et aux types fonctions ($A \multimap B$). L'ensemble des formules est donné inductivement par

$$A, B \stackrel{\text{def}}{=} \alpha \mid A \multimap B \mid !A$$

où α désigne une formule atomique. On présente dans la Figure 6.1 les règles de la Logique Linéaire Intuitionniste en calcul des séquents. Les règles AXIOM, CUT, $\multimap$-RIGHT et $\multimap$-LEFT sont les règles habituelles de la Logique Intuitionniste. Les règles WEAKENING et CONTRACTION sont adaptées à la logique linéaire, c'est à dire qu'on ne peut supprimer ou dupliquer que les termes de la forme $!A$. Enfin, les règles DERELICTION et PROMOTION sont les règles d'introduction à gauche et à droite de l'exponentielle !.

6.1.4 Syntaxe pour la Logique Linéaire Intuitionniste

On souhaite associer à chaque preuve en Logique Linéaire Intuitionniste un terme encodant l'arbre de la preuve. On utilise une version étendue du λ-calcul. On cherche à établir une correspondance, à la CURRY - HOWARD. Par rapport à sa syntaxe, on ajoute les termes $!t$ correspondant au type $!A$. On ajoute aussi le destructeur $\mathsf{let}\, !x = t\, \mathsf{in}\, t$

$$t \stackrel{\text{def}}{=} x \mid \lambda x.t \mid t\ t \mid\ !t \mid \mathsf{let}\, !x = t\, \mathsf{in}\, t$$

On utilise ici le système dont WADLER [Wad93] se sert pour résoudre les problèmes de *cohérence* de la Logique Linéaire (voir Section 6.3). L'ensemble des règles est donné dans

$$\textsc{Axiom}\ \frac{}{x : A \vdash x : A} \qquad \textsc{Cut}\ \frac{\Gamma \vdash t' : B \qquad \Delta, x : B \vdash t : A}{\Delta, \Gamma \vdash t[x/t'] : A}$$

$$\textsc{Contraction}\ \frac{\Gamma, !y : !A, !z : !A \vdash t : B}{\Gamma, !x : !A \vdash t[y/x][z/x] : B} \qquad \textsc{Weakening}\ \frac{\Gamma \vdash t : B}{\Gamma, !x : !A \vdash t : B}$$

$$\multimap\textsc{-Right}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B} \qquad \multimap\textsc{-Left}\ \frac{\Gamma \vdash a : A \qquad \Delta, b : B \vdash t : C}{\Gamma, \Delta, f : A \multimap B \vdash t[b/f\ a] : C}$$

$$\textsc{Dereliction}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma, !x : !A \vdash t : B} \qquad \textsc{Promotion}\ \frac{!\Gamma \vdash t : A}{!\Gamma \vdash !t : !A}$$

$$\textsc{Let}\ \frac{\Gamma, !x : !A \vdash t : B}{\Gamma, y : !A \vdash \mathsf{let}\ !x = y\ \mathsf{in}\ t : B}$$

FIGURE 6.2 – Termes de preuves de la Logique Linéaire Intuitionniste

la Figure 6.2. Un contexte Γ contient des hypothèses usuelles de la forme $x : A$ et des hypothèses $!x : !A$.

$$\Gamma \stackrel{\text{def}}{=} x : A \mid !x : !A$$

On notera $!\Gamma$ lorsque toutes les hypothèses sont de la seconde forme.

6.2 Élimination des coupures

GENTZEN [Gen35] introduit le calcul des séquents et l'élimination des coupures (*Hauptsatz*). La démonstration de ce résultat établit des règles de réécriture des preuves, aboutissant à une preuve sans coupure.

De même, tout séquent $\Gamma \vdash A$ dérivable dans ILL peut l'être sans utiliser la coupure ([Gir87]).

On va montrer ce que les règles de réécriture données dans la démonstration de l'élimination des coupures impliquent sur les règles de réécriture des termes associés aux preuves (Section 6.1.4).

Théorème 6.2.1 (Suppression des coupures)**.** *Si $\Gamma \vdash t : A$ est dérivable dans le système avec coupures, alors il existe t' tel que $\Gamma \vdash t' : A$ est dérivable sans coupure. On obtient les équations suivantes*

$$\begin{aligned} (\lambda x.t)\ t' &= t[x/t'] \\ \mathsf{let}\ !x = !t'\ \mathsf{in}\ t &= t[x/t'] \end{aligned}$$

La démonstration (par induction sur le nombre de règles dans la dérivation) de ce théorème utilise le Lemme 6.2.2 :

Lemme 6.2.2. *Si* $!\Gamma_1 \vdash t : A$ *et* $\Gamma_2, !x : !A \vdash t' : B$ *sont dérivables, alors* $!\Gamma_1, \Gamma_2 \vdash t'[x/t] : B$ *l'est aussi. Autrement dit, la règle suivante est admissible.*

$$\text{MCut}\ \frac{!\Gamma_1 \vdash t : A \qquad \Gamma_2, !x : !A \vdash t' : B}{!\Gamma_1, \Gamma_2 \vdash t'[x/t] : B}$$

Démonstration. La démonstration se fait par induction sur le nombre de règles d'inférence de la preuve de $!\Gamma_2, !x : !A \vdash t' : B$ faisant apparaître la variable de coupure (ici $!x : !A$) du coté gauche de la conclusion (Si une règle (comme la contraction) substitue x par des autres variables, on compte aussi les règles d'inférence faisant apparaître les nouvelles variables que x remplace). Par ailleurs, on notera à chaque fois que si la variable $!z : !C$ apparaît dans Γ_2, et apparaît n fois dans la preuve de $\Gamma_2, !x : !A \vdash t' : B$, elle apparaît encore n fois dans la preuve de $!\Gamma_1, \Gamma_2 \vdash t'[x/t] : B$.

— La règle Axiom n'est pas permise.

— Les règles Cut, $\multimap$-Right et $\multimap$-Left, ainsi que les règles Contraction, Weakening, Dereliction et Promotion lorsque la coupure ne porte pas sur la variable de la partie gauche du séquent sont de simples appels aux hypothèses d'induction. Par exemple, pour la règle $\multimap$-Right, si on a

$$\multimap\text{-Right}\ \frac{\Gamma_2, !x : !A, y : C \vdash t'' : D}{\Gamma_2, !x : !A \vdash \lambda y.t'' : C \multimap D}$$

alors, par hypothèse d'induction, $!\Gamma_1, \Gamma_2, y : C \vdash t''[x/t] : C$. Donc $!\Gamma_1, \Gamma_2 \vdash \lambda y.t''[x/t] : C \multimap D$.

— Dans le cas de la règle Contraction sur $!x : !A$

$$\text{Contraction}\ \frac{\Gamma_2, !y : !A, !z : !A \vdash t'' : B}{\Gamma_2, !x : !A \vdash t''[y/x][z/x] : B}$$

on appelle deux fois l'hypothèse d'induction :

$$\text{MCut}\ \frac{!\Gamma_1 \vdash t : A \qquad \text{Contraction}\ \dfrac{\text{MCut}\ \dfrac{!\Gamma_1 \vdash t : A \qquad \Gamma_2, !y : !A, !z : !A \vdash t'' : B}{!\Gamma_1, \Gamma_2, !z : !A \vdash t''[y/t] : B}}{!\Gamma_1, !\Gamma_1, \Gamma_2 \vdash t''[y/t][z/t] : B}}{!\Gamma_1, \Gamma_2 \vdash t''[y/t][z/t] : B}$$

On notera que les deux appels aux hypothèses d'induction sont bien fondés car si $!x : !A$ apparaît n fois dans la preuve de $\Gamma_2, !x : !A \vdash t''[y/x][z/x] : B$, et que $!y : !A$ et $!z : !A$ apparaissent respectivement m fois et p fois, alors $m + p = n - 1$. Donc $m < n$ et alors le premier appel à MCut

$$\text{MCut}\ \frac{!\Gamma_1 \vdash t : A \qquad \Gamma_2, !y : !A, !z : !A \vdash t'' : B}{!\Gamma_1, \Gamma_2, !z : !A \vdash t''[y/t] : B}$$

est bien fondé. D'autre part, cette preuve ne change pas le nombre d'apparitions de $!z : !A$ (c'est à dire $p < n$). Le second appel à MCut est donc bien fondé également.

— Dans le cas de la règle Weakening sur $!x : !A$

$$\textsc{Weakening}\,\frac{\Gamma_2 \vdash t' : B}{\Gamma_2, !x : !A \vdash t' : B}$$

la règle d'affaiblissement appliquée à $!\Gamma_1$ conclut.

— Dans le cas de la règle Promotion,

$$\textsc{Promotion}\,\frac{!\Gamma'_2, !x : !A \vdash t'' : C}{!\Gamma'_2, !x : !A \vdash !t'' : !C}$$

Par hypothèse d'induction, on a $!\Gamma_1, \Gamma_2 \vdash t''[x/t] : C$, donc $!\Gamma_1, \Gamma_2 \vdash !t''[x/t] : !C$.

— Dans le cas de la règle Dereliction sur $!x : !A$,

$$\textsc{Dereliction}\,\frac{\Gamma_2, x : A \vdash t' : B}{\Gamma_2, !x : !A \vdash t' : B}$$

Par la règle Cut, on a

$$\textsc{Cut}\,\frac{!\Gamma_1 \vdash t : A \qquad \Gamma_2, x : A \vdash t' : B}{!\Gamma_1, \Gamma_2 \vdash t'[x/t] : B}$$

□

La démonstration de la suppression des coupures est habituelle. On va cependant détailler la démonstration dans le cas où la règle à droite de la coupure est la règle Let car la suppression de cette coupure va nous donner la sémantique opérationnelle du let. On s'intéresse donc ici à la coupure

$$\textsc{Cut}\,\frac{\Gamma_1 \vdash t : !A \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\,!x = y\,\mathsf{in}\,t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\,!x = t\,\mathsf{in}\,t' : B}$$

On raisonne selon la dernière règle de la dérivation de $\Gamma_1 \vdash t : !A$. On va voir que seul le cas où cette règle est une promotion a sens calculatoire.

— La règle $\multimap$-Right n'est pas possible.

— Si Dereliction est la dernière règle, la preuve

$$\textsc{Cut}\,\frac{\textsc{Dereliction}\,\dfrac{\Gamma_1, z : C \vdash t : !A}{\Gamma_1, !z : !C \vdash t : !A} \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\,!x = y\,\mathsf{in}\,t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, !z : !C \vdash \mathsf{let}\,!x = t\,\mathsf{in}\,t' : B}$$

se réécrit en

$$\textsc{Cut}\,\frac{\textsc{Dereliction}\,\dfrac{\Gamma_1, z : C \vdash t : !A \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \,\mathsf{in}\, t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, z : C \vdash \mathsf{let}\, !x = t \,\mathsf{in}\, t' : B}}{\Gamma_1, \Gamma_2, !z : !C \vdash \mathsf{let}\, !x = t \,\mathsf{in}\, t' : B}$$

ce qui ne correspond pas à une règle de réécriture des termes de preuve.

— Si $\multimap$-Left est la dernière règle, la preuve

$$\textsc{Cut}\,\frac{\multimap\textsc{-Left}\,\dfrac{\Gamma_1 \vdash c : C \qquad \Gamma_2, d : D \vdash t : !A}{\Gamma_1, \Gamma_2, f : C \multimap D \vdash t[d/f\ c] : !A} \qquad \dfrac{\Gamma_3, !x : !A \vdash t' : B}{\Gamma_3, y : !A \vdash \mathsf{let}\, !x = y \,\mathsf{in}\, t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, \Gamma_3, f : C \multimap D \vdash \mathsf{let}\, !x = t[d/f\ c] \,\mathsf{in}\, t' : B}$$

se réécrit en

$$\multimap\textsc{-Left}\,\frac{\Gamma_1 \vdash c : C \qquad \textsc{Cut}\,\dfrac{\Gamma_2, d : D \vdash t : !A \qquad \dfrac{\Gamma_3, !x : !A \vdash t' : B}{\Gamma_3, y : !A \vdash \mathsf{let}\, !x = y \,\mathsf{in}\, t' : B}\,\textsc{Let}}{\Gamma_2, \Gamma_3, d : D \vdash \mathsf{let}\, !x = t \,\mathsf{in}\, t' : B}}{\Gamma_1, \Gamma_2, \Gamma_3, f : C \multimap D \vdash \mathsf{let}\, !x = t[d/f\ c] \,\mathsf{in}\, t' : B}$$

ce qui ne correspond pas à une règle de réécriture des termes de preuve.

— Si Contraction est la dernière règle, la preuve

$$\textsc{Cut}\,\frac{\textsc{Contraction}\,\dfrac{\Gamma_1, !z_1 : !C, !z_2 : !C \vdash t : !A}{\Gamma_1, !z : !C \vdash t[z_1/z][z_2/z] : !A} \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \,\mathsf{in}\, t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, !z : !C \vdash \mathsf{let}\, !x = t[z_1/z][z_2/z] \,\mathsf{in}\, t' : B}$$

se réécrit en

$$\textsc{Cut}\,\frac{\textsc{Contraction}\,\dfrac{\Gamma_1, !z_1 : !C, !z_2 : !C \vdash t : !A \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \,\mathsf{in}\, t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, !z_1 : !C, !z_2 : !C \vdash \mathsf{let}\, !x = t \,\mathsf{in}\, t' : B}}{\Gamma_1, \Gamma_2, !z : !C \vdash \mathsf{let}\, !x = t[z_1/z][z_2/z] \,\mathsf{in}\, t' : B}$$

ce qui ne correspond pas à une règle de réécriture des termes de preuve.

— Si Weakening est la dernière règle, la preuve

$$\textsc{Cut}\,\frac{\textsc{Weakening}\,\dfrac{\Gamma_1 \vdash t : !A}{\Gamma_1, !z : !C \vdash t : !A} \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \,\mathsf{in}\, t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, !z : !C \vdash \mathsf{let}\, !x = t \,\mathsf{in}\, t' : B}$$

se réécrit en

$$\textsc{Cut}\,\frac{\textsc{Weakening}\,\dfrac{\Gamma_1 \vdash t : !A \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \;\mathsf{in}\; t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\, !x = t \;\mathsf{in}\; t' : B}}{\Gamma_1, \Gamma_2, !z : !C \vdash \mathsf{let}\, !x = t \;\mathsf{in}\; t' : B}$$

ce qui ne correspond pas à une règle de réécriture des termes de preuve.

— Si Cut est la dernière règle, la preuve

$$\textsc{Cut}\,\frac{\textsc{Cut}\,\dfrac{\Gamma_1 \vdash c : C \qquad \Gamma_2, z : C \vdash t : !A}{\Gamma_1, \Gamma_2 \vdash t[z/c] : !A} \qquad \dfrac{\Gamma_3, !x : !A \vdash t' : B}{\Gamma_3, y : !A \vdash \mathsf{let}\, !x = y \;\mathsf{in}\; t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{let}\, !x = t[z/c] \;\mathsf{in}\; t' : B}$$

se réécrit en

$$\textsc{Cut}\,\frac{\Gamma_1 \vdash c : C \qquad \textsc{Cut}\,\dfrac{\Gamma_2, z : C \vdash t : !A \qquad \dfrac{\Gamma_3, !x : !A \vdash t' : B}{\Gamma_3, y : !A \vdash \mathsf{let}\, !x = y \;\mathsf{in}\; t' : B}\,\textsc{Let}}{\Gamma_2, \Gamma_3, z : C \vdash \mathsf{let}\, !x = t \;\mathsf{in}\; t' : B}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{let}\, !x = t[z/c] \;\mathsf{in}\; t' : B}$$

ce qui ne correspond pas à une règle de réécriture des termes de preuve.

— Si Axiom est la dernière règle, la preuve

$$\textsc{Cut}\,\frac{\textsc{Axiom}\,\dfrac{}{z : !A \vdash z : !A} \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \;\mathsf{in}\; t' : B}\,\textsc{Let}}{\Gamma_1, \Gamma_2, z : !A \vdash \mathsf{let}\, !x = z \;\mathsf{in}\; t' : B}$$

se réécrit en

$$\frac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \;\mathsf{in}\; t' : B}\,\textsc{Let}$$

ce qui ne correspond pas à une règle de réécriture des termes de preuve (α-renommage).

— Si Promotion est la dernière règle, la preuve

$$\textsc{Cut}\,\frac{\textsc{Promotion}\,\dfrac{!\Gamma_1 \vdash t : A}{!\Gamma_1 \vdash !t : !A} \qquad \dfrac{\Gamma_2, !x : !A \vdash t' : B}{\Gamma_2, y : !A \vdash \mathsf{let}\, !x = y \;\mathsf{in}\; t' : B}\,\textsc{Let}}{!\Gamma_1, \Gamma_2 \vdash \mathsf{let}\, !x = !t \;\mathsf{in}\; t' : B}$$

se réécrit en une preuve de $!\Gamma_1, \Gamma_2 \vdash t'[x/t] : B$, comme le montre le Lemme 6.2.2. On fera attention de noter que le lemme repousse la règle Cut aux cas où $\Gamma_2, !x : !A \vdash t' : B$ est prouvé par une déréliction. Dans ces cas, le nombre de règles d'inférence est toujours strictement inférieur à celui de la preuve initiale. On peut donc supprimer ces nouvelles règles Cut par hypothèse d'induction.

On a donc la règle

$$(\mathsf{let}\, !x = !t\ \mathsf{in}\ t') = t'[x/t].$$

Remarque 6.2.3. La structure $\mathsf{let}\, !x = t\ \mathsf{in}\ t'$ *bloque* la substitution de t dans t' jusqu'à ce que t ait la forme $!t''$. Cette caractéristique sera essentielle dans notre langage réflexif car elle empêchera de substituer autre chose qu'une donnée dans autre donnée. ■

6.3 Cohérence

6.3.1 Sémantique des preuves

Les modèles usuels de la Logique Linéaire Intuitionniste ont été décrits par Seely dans [See89], corrigés par Bierman [Bie95]. Cette axiomatisation subsume les modèles précédents comme les espaces de cohérences [Gir87] ou la sémantique des jeux [LS91].

D'autres axiomatisations ont été proposées, notamment les des catégories de Lafont [Laf88], les catégories linéaires de Benton, Bierman, de Paiva et Hyland [Ben+92], ou les catégories linéaires-non-linéaires de Benton [Ben95], mais des liens sont possibles entre chacune [Mel03] :

— Une catégorie de Seely est une catégorie linéaire, et une catégorie linéaire avec un produit fini est une catégorie de Seely ([Bie95]).

— Une catégorie linaire induit une catégorie linéaire-non-linéaire ([Ben95]).

— Une catégorie linéaire-non-linéaire définit une catégorie linéaire ([Ben95]).

— Une catégorie de Lafont est une catégorie linéaire ([Bie94]).

Ces axiomatisations sont tous des modèles de la Logique Linéaire Intuitionniste, c'est à dire que pour toute preuve π de $\Gamma \vdash A$, il existe un morphisme $[\![\pi]\!] : \Gamma \to A$, et si la preuve π se réécrit en la preuve π' par suppression des coupures, alors $[\![\pi]\!] = [\![\pi']\!]$.

On donne ici les seuls éléments nécessaires à notre propos, sans rappeler la définition complète d'une catégorie de Seely :

— Une catégorie $\mathbf{C}$ de Seely est symétrique monoïdale : on dispose d'un tenseur $\otimes$, d'une exponentiation $\multimap$, d'une unité I.

— $(\mathbf{C}, \multimap, \mathbf{app}, \lambda())$ est close : $\mathbf{app} : (A \multimap B) \otimes A \to B$ et si $f : A \otimes B \to B$, alors $\lambda(f) : A \to (B \multimap C)$.

— $(\mathbf{C}, \&, 1)$ est cartésienne. On dispose notamment des morphismes (naturels) de duplication $\delta_A : A \to A \,\&\, A$ et d'affaiblissement $\mathbf{discard}_A : A \to 1$.

— $(!, \epsilon, \nu)$ est une comonade : $\epsilon_A : !A \multimap A$ est le morphisme co-unité et $\nu_A : !A \multimap !!A$ est le morphisme de digging. Les diagrammes suivants commutent

$$\textsc{Axiom}\ \frac{}{A \xrightarrow{\mathbf{id}_A} A} \qquad \textsc{Cut}\ \frac{\Gamma \xrightarrow{f} B \qquad \Delta \otimes B \xrightarrow{g} A}{\Delta \otimes \Gamma \xrightarrow{\mathbf{id}_\Delta \otimes f} \Delta \otimes B \xrightarrow{g} A}$$

$$\textsc{Contraction}\ \frac{\Gamma \otimes !A \otimes !A \xrightarrow{f} B}{\Gamma \otimes !A \xrightarrow{\mathbf{id}_\Gamma \otimes ! \delta_A} \Gamma \otimes !A \otimes !A \xrightarrow{f} B} \qquad \textsc{Weakening}\ \frac{\Gamma \xrightarrow{f} B}{\Gamma \otimes !A \xrightarrow{\mathbf{id}_\Gamma \otimes ! \mathbf{discard}_A} \Gamma \xrightarrow{f} B}$$

$$\textsc{Promotion}\ \frac{!\Gamma \xrightarrow{f} A}{!\Gamma \xrightarrow{\nu_\Gamma} !!\Gamma \xrightarrow{!f} !A} \qquad \textsc{Dereliction}\ \frac{\Gamma \otimes A \xrightarrow{f} B}{\Gamma \otimes !A \xrightarrow{\mathbf{id}_\Gamma \otimes \epsilon_A} \Gamma \otimes A \xrightarrow{f} B}$$

$$\multimap\textsc{-Left}\ \frac{\Gamma \xrightarrow{f} A \qquad \Delta \otimes B \xrightarrow{g} C}{\Delta \otimes (A \multimap B) \otimes \Gamma \xrightarrow{\mathbf{id}_{\Delta \otimes (A \multimap B)} \otimes f} \Delta \otimes (A \multimap B) \otimes A \xrightarrow{\mathbf{id}_\Delta \otimes \mathbf{app}} \Delta \otimes B \xrightarrow{g} C} \qquad \multimap\textsc{-Right}\ \frac{\Gamma \otimes A \xrightarrow{f} B}{\Gamma \xrightarrow{\lambda(f)} A \multimap B}$$

FIGURE 6.3 – Sémantique de ILL

$$\begin{array}{ccccc} !A & \xleftarrow{\epsilon_{!A}} & !!A & \xrightarrow{!\,\epsilon_A} & !A \\ & \mathbf{id}_{!A} \searrow & \uparrow \nu_A & \swarrow \mathbf{id}_{!A} & \\ & & !A & & \end{array} \qquad \begin{array}{ccc} !A & \xrightarrow{\nu_A} & !!A \\ {\scriptstyle \nu_A}\downarrow & & \downarrow{\scriptstyle \nu_{!A}} \\ !!A & \xrightarrow[!\,\nu_A]{} & !!!A \end{array}$$

— On dispose des isomorphismes naturels $!1 \simeq I$ et $!(A \,\&\, B) \simeq !A \otimes !B$.

La sémantique d'une preuve de $A_1, \cdots, A_n \vdash B$ est un morphisme de $A_1 \otimes \cdots \otimes A_n \to B$ de $\mathbf{C}$. Ce morphisme est construit inductivement à partir de l'arbre de la preuve. Les règles de construction sont données dans la Figure 6.3.

6.3.2 Problème de la cohérence

On ne demande pas qu'un séquent $\Gamma \vdash t : A$ ne soit associé qu'à une seule preuve, afin d'alléger la syntaxe (par exemple, on ne souhaite pas encoder la règle de coupure dans les termes de preuve). Par contre, on souhaite que toutes les preuves d'un séquent aient la même sémantique. Cette propriété de *cohérence* a été décrite dans [BT+91].

6.3.2.1 Exemple de système non cohérent

Supposons par exemple que le système ne comporte pas d'hypothèse de la forme $!x : !A$, en ayant par exemple les deux règles

$$\frac{\textsc{Promotion}\ \ !\Gamma \vdash t : A}{!\Gamma \vdash !t : !A} \qquad \frac{\textsc{Dereliction}\ \ \Gamma, y : A \vdash t : B}{\Gamma, x : !A \vdash \mathsf{let}\, !y = x\ \mathsf{in}\ t : B}$$

Dans ce cas, le système n'est pas cohérent. Posons $\mathbf{prom}(f) \stackrel{\text{def}}{=} !f \circ \nu$. La règle sémantique de la promotion de la Figure 6.3 peut donc se réécrire sous la forme

$$\frac{\textsc{Promotion}\ \ !\Gamma \xrightarrow{f} A}{!\Gamma \xrightarrow{\mathbf{prom}(f)} !A}$$

Comme noté par Wadler dans [Wad92], le séquent $x : !!A \vdash !\,\mathsf{let}\, !y = x\ \mathsf{in}\ y : !!A$ a la dérivation

$$\dfrac{\textsc{Promotion}\ \dfrac{\textsc{Dereliction}\ \dfrac{\textsc{Axiom}\ \dfrac{}{y : !A \vdash y : !A}}{x : !!A \vdash \mathsf{let}\, !y = x\ \mathsf{in}\ y : !A}}{x : !!A \vdash !\,\mathsf{let}\, !y = x\ \mathsf{in}\ y : !!A}}{}$$

qui a la sémantique $\mathbf{prom}(\mathbf{id}_{!A} \circ \epsilon_{!A}) = !(\mathbf{id}_{!A} \circ \epsilon_{!A}) \circ \nu_{!A} = \mathbf{id}_{!!A}$ ou encore la dérivation

$$\dfrac{\textsc{Cut}\ \dfrac{\textsc{Dereliction}\ \dfrac{\textsc{Axiom}\ \dfrac{}{y : !A \vdash y : !A}}{x : !!A \vdash \mathsf{let}\, !y = x\ \mathsf{in}\ y : !A} \qquad \dfrac{\dfrac{}{z : !A \vdash z : !A}\ \textsc{Axiom}}{z : !A \vdash !z : !!A}\ \textsc{Promotion}}{x : !!A \vdash !\,\mathsf{let}\, !y = x\ \mathsf{in}\ y : !!A}}{}$$

qui a la sémantique $\mathbf{prom}(\mathbf{id}_{!A}) \circ \epsilon_{!A} = !\,\mathbf{id}_{!A} \circ \nu_A \circ \epsilon_{!A} = \nu_A \circ \epsilon_{!A}$ qui n'est pas égale à $\mathbf{id}_{!!A}$ dans le cas général.

6.3.2.2 Analyse du problème de cohérence

Le problème de la cohérence vient du fait que $\mathbf{prom}(f) \circ g = \mathbf{prom}(f \circ g)$ n'est pas vraie dans le cas général. Or, cette égalité est nécessaire lors de l'utilisation de la règle de coupure.

Les solutions adoptées ([Ben+92 ; Wad93]) utilisent la même technique qui est d'interdire toute coupure sur une variable libre d'un terme issu d'une promotion (de la forme $!t$ dans notre syntaxe). Chez Wadler, cette solution prend la forme d'un séquent modifié

$$!\Gamma_1, \Gamma_2 \vdash t : B$$

où $!\Gamma_1$ est de la forme $!x_1 : !A_1, \cdots, !x_n : !A_n$ et Γ_2 est de la forme $y_1 : B_1, \cdots, y_m : B_m$ et d'une règle Cut de la forme

$$\frac{\textsc{Cut}\ \ \Gamma \vdash t' : B \qquad \Delta, x : B \vdash t : A}{\Delta, \Gamma \vdash t[x/t'] : A}$$

L'ensemble de variables Γ_2 désigne donc l'ensemble des formules pouvant être branchées à un Cut. Les formules de $!\Gamma_1$, elles, ne le peuvent pas. La promotion devient alors

$$\text{Promotion}\ \frac{!\Gamma \vdash t : B}{!\Gamma \vdash !t :!B}$$

afin d'empêcher les branchements à des règles Cut. La déréliction se décompose à présent en deux règles

$$\text{Let}\ \frac{\Gamma, !y :!A_i \vdash t : B}{\Gamma, x :!A_i \vdash \mathsf{let}\, !y = x\ \mathsf{in}\ t : B} \qquad \text{Dereliction}\ \frac{\Gamma, x : A_i \vdash t : B}{\Gamma, !x :!A_i \vdash t : B}.$$

La règle Let n'a pas de sens logique. Elle sert uniquement à faire passer une formule $!A$ de son état $x : !A$ à son état $!x : !A$. Cette syntaxe de la logique linéaire intuitionniste est cohérente [Wad93].

Si l'égalité $\mathbf{prom}(f \circ g) = \mathbf{prom}(f) \circ g$ est fausse en générale, elle est valide lorsque g est de la forme $\mathbf{prom}(g')$. Il serait donc possible de couper sur une variable libre d'un terme $!t$ si le terme branché dans la coupure était lui même de la forme $!t'$. C'est l'intérêt de l'admissibilité de la coupure du Lemme 6.2.2 : si un jugement $!\Delta \vdash !t : !A$ est dérivable et que $\Gamma, !x : !A \vdash t' : B$ l'est aussi, on peut couper sur x, même si, dans ce cas, x peut être une variable libre dans un terme $!(\cdots)$.

6.4 Soft Linear Logic

6.4.1 Preuves en SLL

La Soft Linear Logic est un sous-système strict de la Logique Linéaire Intuitionniste. Elle a été introduite par Lafont [Laf02] pour caractériser les fonctions dans PTIME. Nous ne l'utiliserons ici que comme un sous-système strict de la logique linéaire, pour mettre en évidence les différentes interprétations que peuvent avoir les règles exponentielles (Dereliction et Promotion/SoftPromotion). Elle est définie par les règles de la Logique Linéaire Intuitionniste où les règles Contraction, Weakening, Dereliction et Promotion on été remplacées par les règles suivantes.

$$\text{SoftPromotion}\ \frac{\Gamma \vdash A}{!\Gamma \vdash !A} \qquad \text{Multiplexing}\ \frac{\Gamma, A^{(n)} \vdash B}{\Gamma, !A \vdash B}$$

La règle Multiplexing est valable pour tout entier naturel n. Pour $n = 0$, $n = 1$, on retrouve notamment les règles Weakening, Dereliction. On peut retrouver cette règle avec ces deux règles additionnées de la règle Contraction. En effet, on peut retrouver la règle Multiplexing pour $n > 1$ par composition successive de la règle Contraction et de règle Dereliction. Par exemple, le cas $n = 2$ peut se traiter de la manière suivante

$$\begin{array}{r} \text{Contraction}\ \dfrac{\text{Dereliction}\ \dfrac{\text{Dereliction}\ \dfrac{\Gamma, A, A \vdash B}{\Gamma, A, !A \vdash B}}{\Gamma, !A, !A \vdash B}}{\Gamma, !A \vdash B} \end{array}$$

La règle SoftPromotion peut elle être retrouvée à partir des règles Promotion et Dereliction. Par exemple, si $!\Gamma = !A, !B, !C$ alors on a

$$\dfrac{\dfrac{\dfrac{\dfrac{A, B, C \vdash D}{A, B, !C \vdash D}\,\textsc{Dereliction}}{A, !B, !C \vdash D}\,\textsc{Dereliction}}{!A, !B, !C \vdash D}\,\textsc{Dereliction}}{!A, !B, !C \vdash !D}\,\textsc{Promotion}$$

Toutes les formules prouvables en Logique Linéaire sont donc prouvables en Soft Linear Logic.

6.4.2 De SLL à LL

On retrouve toute l'expressivité de la logique linéaire en ajoutant aux règles SoftPromotion et Multiplexing la règle Digging

$$\dfrac{\Gamma, !!A \vdash B}{\Gamma, !A \vdash B}\,\textsc{Digging}$$

En effet, les règles Promotion et Contraction peuvent être écrites de la manière suivante

$$\dfrac{\dfrac{\dfrac{\dfrac{!A, !B, !C \vdash D}{!!A, !!B, !!C \vdash !D}\,\textsc{SoftPromotion}}{!!A, !!B, !C \vdash !D}\,\textsc{Digging}}{!!A, !B, !C \vdash !D}\,\textsc{Digging}}{!A, !B, !C \vdash !D}\,\textsc{Digging}$$

$$\dfrac{\dfrac{\Gamma, !A, !A \vdash B}{\Gamma, !!A \vdash B}\,\textsc{Multiplexing}}{\Gamma, !A \vdash B}\,\textsc{Digging}$$

Et le système avec les règles SoftPromotion, Digging et Multiplexing est compris dans Logique Linéaire Intuitionniste car la règle Digging peut être retrouvée de la manière suivante.

$$\dfrac{\dfrac{\dfrac{}{!A \vdash !A}\,\textsc{Axiom}}{!A \vdash !!A}\,\textsc{Promotion} \qquad \Gamma, !!A \vdash B}{\Gamma, !A \vdash B}\,\textsc{Cut}$$

Une formule est prouvable en Logique Linéaire si et seulement si elle est prouvable dans SLL + Digging.

6.4.3 Assignation de termes

La question d'une syntaxe en λ-calcul correcte (au sens de la réduction du sujet) pour SLL a été traitée par [BM04 ; GRR07]. Dans le deuxième article il est rappelé l'importance de ne faire porter les coupures que sur des types linéaires (qui ne vont pas être dupliqués ou effacés). Pour cela, le système de types explicite que la coupure ne peut se faire que sur des types qui ne sont pas de la forme $!A$. Le système de Wadler utilise une autre stratégie, qui est de faire porter les coupures uniquement sur les variables de la forme $x : A$, c'est à dire sur les variables linéaires (les seules variables que l'on peut dupliquer ou supprimer sont celles de la forme $!x : !A$).

6.5 Conclusion

La Logique Linéaire introduit la modalité exponentielle. Dans le cadre de la gestion des ressources, elle décrit les formules que l'on peut utiliser à loisir. Elle pose le problème de la cohérence de la syntaxe associée à cette logique. En effet, la promotion ne commute pas avec la coupure en général. Cependant, elle commute avec la coupure dans le cas où la preuve branchée est elle même terminée par une promotion.

Une solution au problème de cohérence est donc d'interdire les coupures sur les variables libres des termes de promotion en général, et de permettre la coupure lorsque l'on peut vérifier que la preuve branchée est elle même une promotion.

Ce problème sémantique ressemble de manière surprenante au problème de confluence du Chapitre 7, aussi bien dans sa formulation que dans sa résolution.

Chapitre 7

Un langage réflexif

On présente Λ_R, un langage de programmation fonctionnelle inspiré par le λ-calcul, qui a la capacité de geler l'exécution d'un terme. Il correspond à l'idée que l'on se fait d'une donnée, car il ne peut être réduit (comme une donnée ne peut être exécutée). De plus, ce terme gelé n'est pas clos et on peut substituer ses variables libres par d'autres termes gelés. On montre que notre langage est confluent, grâce à une gestion fine des variables libres *permettant la substitution dans les termes gelés uniquement avec d'autres termes gelés.*

7.1 Définition du langage

7.1.1 Principe

Notre langage est une extension du λ-calcul qui permet de stopper la réduction d'un terme. On note ces termes $\langle t \rangle$. On dira que le terme t est *gelé*. Les termes de notre langage sont les éléments t de l'ensemble Λ_R défini inductivement :

$$\Lambda_R \ni t \stackrel{\text{def}}{=} x \mid t\ t \mid \lambda x.t \mid \langle t \rangle \mid \mathsf{let}\ \langle x \rangle = t\ \mathsf{in}\ t$$

Ici, la structure $\langle\rangle$ reste syntaxique. Elle ne correspond pas à un un encodage particulier, ou dit autrement, l'encodage est laissé indéfini. Le processus de réflexion (au sens de la section 2.3) (qui passe de $\langle t \rangle$ à t) est implicitement en charge du désencodage. Un exemple historique d'un tel encodage est donné par la numérotation de Gödel. Mais on peut aussi le définir sur un ensemble $\mathcal{D}$ plus expressif, comme par exemple dans [Jon97] : l'encodage de t peut être l'arbre de la syntaxe (AST) de t dans le domaine des listes, ou une chaîne de caractères contenant un chiffré de cet AST.

Il est possible d'effectuer certaines substitutions dans les termes gelés : on peut substituer des termes gelés dans d'autres termes gelés. Mais tout autre substitution est interdite. En effet, si n'importe quelle substitution était permise, la confluence pourrait être perdue. Par exemple, supposons que l'on puisse substituer n'importe quelle terme dans $\langle x \rangle$. Dans ce cas, $(\lambda x.\langle x \rangle)((\lambda x.x)\lambda x.x)$ se réduirait à la fois en $\langle(\lambda x.x)\lambda x.x\rangle$ et en $\langle \lambda x.x \rangle$, qui sont deux formes normales différentes. Notre solution consiste à ne permettre la substitution à travers les chevrons $\langle\rangle$ que de termes eux mêmes gelés.

Il est intéressant de noter que le problème de substitution dans les termes gelés est très similaire à celui rencontré par WADLER essayant de donner une syntaxe à la logique linéaire, comme décrit dans la Section 6.3. Le problème de cohérence de la syntaxe de la Logique Linéaire trouve ici une interprétation en terme de confluence dans un système utilisant la notion de donnée.

7.1.2 Contextes

De la même manière que dans la Section 5.1.2, on définit les *contextes de structure*

$$S \stackrel{\text{def}}{=} [] \mid S\ t \mid t\ S \mid \lambda x.S \mid \langle S \rangle \mid \mathsf{let}\ \langle x \rangle = S\ \mathsf{in}\ t \mid \mathsf{let}\ \langle x \rangle = t\ \mathsf{in}\ S$$

Un *contexte de donnée* D est un contexte de structure de la forme $S_1 \circ \langle S_2 \rangle$. On peut donc écrire

$$D \stackrel{\text{def}}{=} \langle S \rangle \mid D\ t \mid t\ D \mid \lambda x.D \mid \mathsf{let}\ \langle x \rangle = D\ \mathsf{in}\ t \mid \mathsf{let}\ \langle x \rangle = t\ \mathsf{in}\ D$$

Un *contexte de réduction* E est un contexte de structure qui n'est pas un contexte de donnée.

Propriété 7.1.1. *L'ensemble des contextes de réduction peut être défini inductivement par*

$$E \stackrel{def}{=} [] \mid E\ t \mid t\ E \mid \lambda x.E \mid \mathsf{let}\ \langle x \rangle = E\ \mathsf{in}\ t \mid \mathsf{let}\ \langle x \rangle = t\ \mathsf{in}\ E$$

Démonstration. Posons A l'ensemble des contextes de réduction (c'est à dire les contextes de structure qui ne sont pas des contextes de donnée) et B l'ensemble défini inductivement dans la propriété. Montrons que $A = B$.

D'une part, si $S \in A$, alors, on peut montrer par induction sur S que $S \in B$.

D'autre part, si $S \in B$, alors S n'est pas un contexte de données (car, pour tout S, $\langle S \rangle$ ne fait pas parti des contextes de B). □

On dira qu'un terme t' est *gelé* dans t s'il existe une contexte de donnée D tel que $t = D[t']$. Le terme t' est donc gelé s'il est entouré par $\langle \cdots \rangle$ dans t.

7.1.3 Liaison des variables

La sémantique opérationnelle du langage est un peu délicate du fait du risque de captures de variables non souhaitées. Comme on l'a dit dans l'introduction de la section, les substitutions à travers les chevrons $\langle\rangle$ ne peuvent pas être faites dans n'importe quelle situation. La solution choisie reprend celle de WADLER dans [Wad93], qui consiste à ne permettre la liaison les variables libres de $\langle t \rangle$ qu'avec un lieur spécial, différent de λ, afin de n'autoriser les substitutions dans $\langle t \rangle$ que de termes de la forme $\langle t' \rangle$. Par exemple, la variable x devrait être libre dans le terme $\lambda x.\lambda z.z\ \langle \lambda y.xy \rangle$ alors que y et z sont liés (en fait, il est α-équivalent à $\lambda x'.\lambda z.z\ \langle \lambda y.xy \rangle$). Le terme $(\lambda x.\langle x \rangle)t$ se réduit donc en $\langle x \rangle$ car l'occurrence de x dans $\langle x \rangle$ est libre dans $\lambda x.\langle x \rangle$. Le lieur spécial qui permet de lier une variable à travers $\langle\rangle$ est let.

La notion de variable libre est donc un peu différente de celle en λ-calcul. Il est à noter que le lieur let (contrairement au lieur λ) peut lier une variable libre *partout* dans le terme, que cette variable soit ou non entourée par $\langle\rangle$.

Exemple 7.1.2. La variable libre x de $\langle x \rangle$ n'est pas liée à λx dans le terme $\lambda x.\langle x \rangle$. Elle l'est par contre par $\mathsf{let}\,\langle x \rangle$ dans le terme $\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ \langle x \rangle$. *A contrario*, la variable libre x dans $\lambda y.x$ est liée à $\mathsf{let}\,\langle x \rangle$ dans le terme $\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ \lambda y.x$. ■

On pose donc $\mathrm{FV}^{\lambda}(t)$ l'ensemble des variables libres pouvant être liées par λ (c'est à dire les variables libres qui ne sont pas entourées par $\langle\rangle$) et $\mathrm{FV}^{\mathsf{let}}(t)$ l'ensemble des variables libres ne pouvant être liées que par let (c'est à dire les variables libres des termes entourées par $\langle\rangle$).

$$
\begin{aligned}
\mathrm{FV}^{\lambda}(x) &\stackrel{\text{def}}{=} \{x\} \\
\mathrm{FV}^{\lambda}(\lambda x.t) &\stackrel{\text{def}}{=} \mathrm{FV}^{\lambda}(t) \setminus x \\
\mathrm{FV}^{\lambda}(t_1\ t_2) &\stackrel{\text{def}}{=} \mathrm{FV}^{\lambda}(t_1) \cup \mathrm{FV}^{\lambda}(t_2) \\
\mathrm{FV}^{\lambda}(\langle t \rangle) &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{FV}^{\lambda}(\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ t') &\stackrel{\text{def}}{=} \mathrm{FV}^{\lambda}(t) \cup (\mathrm{FV}^{\lambda}(t') \setminus x)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{FV}^{\mathsf{let}}(x) &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{FV}^{\mathsf{let}}(\lambda x.t) &\stackrel{\text{def}}{=} \mathrm{FV}^{\mathsf{let}}(t) \\
\mathrm{FV}^{\mathsf{let}}(t_1\ t_2) &\stackrel{\text{def}}{=} \mathrm{FV}^{\mathsf{let}}(t_1) \cup \mathrm{FV}^{\mathsf{let}}(t_2) \\
\mathrm{FV}^{\mathsf{let}}(\langle t \rangle) &\stackrel{\text{def}}{=} \mathrm{FV}^{\lambda}(t) \cup \mathrm{FV}^{\mathsf{let}}(t) \\
\mathrm{FV}^{\mathsf{let}}(\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ t') &\stackrel{\text{def}}{=} \mathrm{FV}^{\mathsf{let}}(t) \cup (\mathrm{FV}^{\mathsf{let}}(t') \setminus x)
\end{aligned}
$$

L'ensemble des variables libres FV de t est l'*union* de $\mathrm{FV}^{\lambda}(t)$ et $\mathrm{FV}^{\mathsf{let}}(t)$. On notera que $\mathrm{FV}^{\lambda}(\langle t \rangle) = \emptyset$ et $\mathrm{FV}^{\mathsf{let}}(\langle t \rangle) = \mathrm{FV}(t)$, car aucune variable libre de t ne peut être liée par un lieur λ, mais elles le sont toutes par un lieur let. De plus, un lieur let peut lier toutes les variables, gelées ou pas. Donc on doit retirer x de $\mathrm{FV}^{\lambda}(t')$ (resp. $\mathrm{FV}^{\mathsf{let}}(t')$) dans le calcul de $\mathrm{FV}^{\lambda}(\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ t')$ (resp. $\mathrm{FV}^{\mathsf{let}}(\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ t')$).

Exemple 7.1.3. On a les résultats suivants :

- $\mathrm{FV}^{\lambda}(\lambda x.\langle x \rangle) = \emptyset$ et $\mathrm{FV}^{\mathsf{let}}(\lambda x.\langle x \rangle) = \{x\}$
- $\mathrm{FV}(\mathsf{let}\,\langle x \rangle = \langle \lambda y.y \rangle\ \mathsf{in}\ \langle x \rangle) = \emptyset$
- $\mathrm{FV}^{\lambda}(\lambda x.f\langle x\langle \lambda y.y \rangle\rangle) = \{f\}$ et $\mathrm{FV}^{\mathsf{let}}(\lambda x.f\langle x\langle \lambda y.y \rangle\rangle) = \{x\}$
- $\mathrm{FV}^{\lambda}(\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ x) = \mathrm{FV}^{\lambda}(t)$ et $\mathrm{FV}^{\mathsf{let}}(\mathsf{let}\,\langle x \rangle = t\ \mathsf{in}\ x) = \mathrm{FV}^{\mathsf{let}}(t)$

■

7.1.4 α-renommage

À partir de maintenant, on considère les termes modulo α-renommage : les variables liées seront renommées de sorte à ce qu'elles soient toujours différentes des variables libres.

Exemple 7.1.4. $\mathsf{let}\,\langle x \rangle = z\ \mathsf{in}\ \lambda x.x\langle x \rangle$ est α-équivalent à $\mathsf{let}\,\langle y \rangle = z\ \mathsf{in}\ \lambda x.x\langle y \rangle$. ■

7.1.5 Substitution

Les règles de substitution sont données par

$$\begin{array}{rcll} x[x/t'] & = & t' & \\ y[x/t'] & = & y & x \neq y \\ (\lambda y.t)[x/t'] & = & \lambda y.t[x/t'] & x \neq y \text{ and } y \notin \mathrm{FV}(t') \\ (t_1 t_2)[x/t'] & = & t_1[x/t']t_2[x/t'] & \\ \langle t \rangle[x/t'] & = & \langle t[x/t'] \rangle & \\ (\mathsf{let}\ \langle y \rangle = t_1\ \mathsf{in}\ t_2)[x/t'] & = & \mathsf{let}\ \langle y \rangle = t_1[x/t']\ \mathsf{in}\ t_2[x/t'] & x \neq y \text{ and } y \notin \mathrm{FV}(t') \end{array}$$

7.1.6 Redexes et β-réduction

Les règles de réduction s'appuient sur les contractions des λ-redexes et des let-redexes.

$$\begin{array}{rcl} E[(\lambda x.t')\ t] & \stackrel{\lambda}{\to} & E[t'[x/t]] \\ E[\mathsf{let}\ \langle x \rangle = \langle t \rangle\ \mathsf{in}\ t'] & \stackrel{\mathsf{let}}{\to} & E[t'[x/t]] \end{array}$$

où E est un contexte de réduction. Finalement, on définit $\to \stackrel{\text{def}}{=} \stackrel{\lambda}{\to} \cup \stackrel{\mathsf{let}}{\to}$ La clôture réflexive transitive de $\to$ est notée $\stackrel{*}{\to}$. On note également $\to_{\text{head}}$ la restriction de $\to$ lorsque $E = []$.

Il est important de noter que $\langle E \rangle$ n'est jamais un contexte de réduction. Il n'est donc pas possible de réduire un terme entouré par $\langle\rangle$. Ce comportement est attendu pour une donnée. Lorsqu'un redex peut être atteint par un contexte E, on dit que ce redex est *actif*. Autrement, c'est un *redex inactif*.

Le lieur let permet de substituer des termes gelés dans d'autres termes gelés. Par exemple, observons une réduction possible faisant intervenir ce lieur :

$$\begin{array}{rl} & \mathsf{let}\ \langle x \rangle = \underline{(\lambda x.x)\langle s \rangle}\ \mathsf{in}\ (\lambda xy.y)\ \langle (\lambda y.y)x \rangle\ \langle x \rangle \\ \to & \underline{\mathsf{let}\ \langle x \rangle = \langle s \rangle\ \mathsf{in}\ (\lambda xy.y)\ \langle (\lambda y.y)x \rangle\ \langle x \rangle} \\ \to & \underline{(\lambda xy.y)\ \langle (\lambda y.y)s \rangle\ \langle s \rangle} \\ \to & \langle s \rangle \end{array}$$

Le calcul s'arrête ici car $\langle s \rangle$ est gelé. Il est à noter que la réduction du let est bloquée jusqu'à ce que $(\lambda x.x)\langle s \rangle$ se réduise en $\langle s \rangle$.

Il est aussi possible de supprimer les parenthèses $\langle\rangle$ autour d'un terme gelé pour permettre son exécution. Le terme $\mathsf{run} \stackrel{\text{def}}{=} (\lambda x.\, \mathsf{let}\ \langle y \rangle = x\ \mathsf{in}\ y)$ en est un exemple typique. Par exemple,

$$\begin{array}{rl} & \mathsf{run}\ \langle t \rangle \\ \to & \underline{(\lambda x.\, \mathsf{let}\ \langle y \rangle = x\ \mathsf{in}\ y)\langle t \rangle} \\ \to & \underline{\mathsf{let}\ \langle y \rangle = \langle t \rangle\ \mathsf{in}\ y} \\ \to & t \end{array}$$

Le terme run a donc permis de supprimer les parenthèses $\langle \cdots \rangle$ autour du terme t, lui permettant alors de se réduire. On a donc transformé un calcul suspendu (une donnée) en un calcul actif (programme). On a donc *réflechi* une donnée.

Remarque 7.1.5. Pour plus de concision, on pourra noter $\lambda\langle x \rangle.t$ le terme $\lambda X.\, \mathsf{let}\ \langle x \rangle = X\ \mathsf{in}\ t$. Par exemple, le terme run peut donc s'écrire $\lambda\langle x \rangle.x$. ■

7.2 Confluence

On va maintenant prouver que le système $(\Lambda_R, \to)$ est confluent. Le point clé pour comprendre ce résultat est que les variables libres de $\langle t \rangle$ ne sont pas capturées par les lieurs λ. Le terme $(\lambda x.\, \mathsf{let}\, \langle y \rangle = x\, \mathsf{in}\, \langle y \rangle)((\lambda x.\langle a \rangle)\ \lambda x.x)$ se réduit à $\langle a \rangle$ car la construction let force l'évaluation de son premier argument $(\lambda x.\langle a \rangle)\lambda x.x$, et seulement ensuite contracte le let-redex.

Théorème 7.2.1. *$(\Lambda_R, \to)$ est confluent, c'est à dire, si $t \xrightarrow{*} t'$ et $t \xrightarrow{*} t''$ alors il existe un terme s tel que $t' \xrightarrow{*} s$ et $t'' \xrightarrow{*} s$.*

Pour montrer la confluence de $(\Lambda_R, \to)$, on le décompose en deux systèmes $(\Lambda_R, \xrightarrow{\lambda})$ et $(\Lambda_R, \xrightarrow{\mathsf{let}})$. On prouve ensuite, de manière séparée que ces deux sous-systèmes sont confluents et que les deux systèmes *commutent*, c'est à dire que si $t \xrightarrow{*\lambda} t_1$ et $t \xrightarrow{*\mathsf{let}} t_2$, alors il existe s tel que $t_1 \xrightarrow{*\mathsf{let}} s$ et $t_2 \xrightarrow{*\lambda} s$. Par le Lemme de Hindley-Rosen [Ros73 ; Hin64], on conclut que $(\Lambda_R, \to) = (\Lambda_R, \xrightarrow{\lambda} \cup \xrightarrow{\mathsf{let}})$ est confluent.

Propriété 7.2.2. *Le système $(\Lambda_R, \xrightarrow{\lambda})$ est confluent.*

Démonstration. Le lieur λ ne peut pas lier de variables libres de $\langle \cdots \rangle$. Donc, sans règle $\xrightarrow{\mathsf{let}}$ les termes entourés par $\langle\rangle$ ne sont pas modifiables. Le système $(\Lambda_R, \xrightarrow{\lambda})$ hérite donc de la confluence du λ-calcul avec constantes (chaque terme $\langle t \rangle$ est une constante) et produit cartésien (chaque terme $\mathsf{let}\, \langle x \rangle = t\, \mathsf{in}\, t'$ est encodé par $(t, \lambda x.t')$). □

Propriété 7.2.3. *Si $t \xrightarrow{\mathsf{let}} t'$ et $t \xrightarrow{\mathsf{let}} t''$ alors il existe un terme s tel que $t' \xrightarrow{\mathsf{let}} s$ et $t'' \xrightarrow{\mathsf{let}} s$ (confluence forte). On en déduit que $(\Lambda_R, \xrightarrow{\mathsf{let}})$ est confluent.*

Démonstration. Soit le terme P contenant deux let-redexes R et S. Supposons que $R = \mathsf{let}\, \langle x \rangle = \langle t \rangle\, \mathsf{in}\, t'$ et qu'on contracte R. S a un unique redex résiduel par la contraction de R. En effet, supposons que S soit un sous-terme de R. S ne peut être un sous terme de $\langle t \rangle$. Donc S est un sous terme de t'. On en déduit que S n'est pas dupliqué. De même, si S n'est pas un sous-terme de R, alors S a toujours un unique redex résiduel par la contraction de R.

Comme il n'y a pas de duplication de S par R (et par symétrie, de R par S), on peut contracter indifféremment S et R. □

Propriété 7.2.4. *Les règles $\xrightarrow{\lambda}$ et $\xrightarrow{\mathsf{let}}$ commutent.*

Démonstration. Soient R et S un λ-redex et un let-redex. On suppose que les deux redexes sont actifs, c'est à dire qu'il ne sont pas dans des termes gelés. La contraction de S ne duplique pas de R. En effet, si S est de la forme $\mathsf{let}\, \langle x \rangle = \langle t \rangle\, \mathsf{in}\, t'$, alors R ne peut se trouver dans $\langle t \rangle$, car R est un redex actif. D'autre part, la contraction de R peut créer différentes copies de S. Donc, pour tout t, t_1, t_2 tel que $t \xrightarrow{\lambda} t_1$ et $t \xrightarrow{\mathsf{let}} t_2$, il existe s tel que $t_1 \xrightarrow{*\mathsf{let}} s$ et $t_2 \xrightarrow{\lambda} s$.

On en déduit que si $t \xrightarrow{\lambda} t_1$ et $t \xrightarrow{*\mathsf{let}} t_2$, alors il existe s tel que $t_2 \xrightarrow{\lambda} s$ et $t_1 \xrightarrow{*\mathsf{let}} s$ (par induction sur la longueur de $t \xrightarrow{*\mathsf{let}} t_2$).

On en déduit que si $t \xrightarrow{*\lambda} t_1$ et $t \xrightarrow{*\mathsf{let}} t_2$, alors il existe s tel que $t_1 \xrightarrow{*\mathsf{let}} s$ et $t_2 \xrightarrow{*\lambda} s$ (par induction sur la longueur de $t \xrightarrow{*\lambda} t_1$). □

7.3 Réflexion

Des deux notions de *réflexion* et de *réification*, seule la première trouve une implémentation facile dans ce langage : il est facile de construire une terme run prenant une donnée $\langle t \rangle$ et lui associant le programme t. Le problème de la réification est plus difficile à résoudre. Des solutions seront présentées dans le Chapitres 11.

7.3.1 Réflexion

Les termes gelés $\langle t \rangle$ correspondent aux données. On a vu qu'il était possible de supprimer les chevrons $\langle\rangle$ en utilisant le terme run. Cet terme agit donc à la manière d'un *interpréteur* (Section 1.3.3).

Une tour de réflexion est, par définition, la spécialisation d'un interpréteur récursif à une donnée (un programme réifié). Si on utilise simplement un interpréteur à la place, comme dans le cas d'une tour de réflexions sans réification (Section 2.4.3.3), on peut écrire une tour de taille n dans Λ_R de la manière suivante

$$\forall i \in [\![0, n-1]\!] \quad p_i = \mathsf{run}\ \langle p_{i+1} \rangle$$

7.3.2 Problème de la réification

Comme on l'a vu dans la Partie 2.4, le cadre d'étude de Smith décompose les phénomènes réflexifs en deux comportements : la réflexion (c'est à dire la capacité d'exécuter une donnée) et la réification (transformer un programme en donnée). Le langage que nous avons présenté ici répond bien à la question de la réflexion. Les données étant de la forme $\langle t \rangle$, il existe un terme du langage, le terme $\mathsf{run} = \lambda x.\, \mathsf{let}\, \langle y \rangle = x\ \mathsf{in}\ y$, qui permet de passer du terme gelé $\langle t \rangle$ au terme exécutable t.

Par contre, la question de la réification est plus difficile à résoudre. Le résultat suivant montre qu'il n'existe pas de terme de notre langage permettant d'associer à un terme t son terme gelé $\langle t \rangle$

Propriété 7.3.1. *Sachant que $(\Lambda_R, \to)$ est confluent, il n'existe pas de terme $\mathsf{box} \in \Lambda_R$ tel que, pour tout terme $t \in \Lambda_R$, $\mathsf{box}\ t \xrightarrow{*} \langle t \rangle$.*

Démonstration. Par l'absurde. Si box existait, alors $\mathsf{box}\ (\lambda x.x)\ \lambda x.x \xrightarrow{*} \langle (\lambda x.x)\ \lambda x.x \rangle$ et $\mathsf{box}\ (\lambda x.x)\ \lambda x.x \to \mathsf{box}\ \ \lambda x.x \xrightarrow{*} \langle \lambda x.x \rangle$, ce qui est impossible car les deux sont en formes normales différentes alors que le langage est confluent. □

7.4 Adhésion au modèle des langages de programmation

7.4.1 Définition du langage

Dans cette partie, on décrit le langage Λ_R dans le cadre des langages de programmation décrit dans la Section 1.2. On pose $\mathcal{P} \overset{\text{def}}{=} \mathcal{D} \overset{\text{def}}{=} \Lambda_R$.

Données On note $\mathcal{D}^*$ l'ensemble des listes de données. Si $\mathbf{d} = [d_1, \cdots, d_n] \in \mathcal{D}^*$, on note $\lfloor \mathbf{d} \rfloor$ le terme $\lambda x.x\ \langle d_1 \rangle\ \cdots\ \langle d_n \rangle$. On note que pour tout $\mathbf{d}$, $\lfloor \mathbf{d} \rfloor$ est en forme normale.

Programmes Si $p \in \mathcal{P}$, la sémantique $[\![p]\!]\,(\mathbf{d}) = \mathbf{d}'$ lorsque $\lambda x.\lfloor \mathbf{d} \rfloor (p\ x)$ a comme forme normale $\lfloor \mathbf{d}' \rfloor$, ou alors, ce qui est équivalent, lorsque $\lambda x.\lfloor \mathbf{d} \rfloor (p\ x) =_\beta \lfloor \mathbf{d}' \rfloor$.

Exemple 7.4.1. Le terme $\mathsf{id} \stackrel{\text{def}}{=} \lambda x.x$ vérifie$[\![\mathsf{id}]\!]\,(d.\mathbf{d}) = d.\mathbf{d}$ ■

Exemple 7.4.2. Le terme $\mathsf{dupl} \stackrel{\text{def}}{=} \lambda x.\lambda a.x\ a\ a$ vérifie $[\![\mathsf{dupl}]\!]\,(d.\mathbf{d}) = d.d.\mathbf{d}$ ■

Exemple 7.4.3. Le terme $\mathsf{pop} \stackrel{\text{def}}{=} \lambda x.\lambda a.x$ vérifie $[\![\mathsf{pop}]\!]\,(d.\mathbf{d}) = \mathbf{d}$ ■

Exemple 7.4.4. Le terme $\mathsf{switch} \stackrel{\text{def}}{=} \lambda x.\lambda a.\lambda b.x\ b\ a$ vérifie $[\![\mathsf{switch}]\!]\,(d_1.d_2.\mathbf{d}) = d_2.d_1.\mathbf{d}$ ■

7.4.2 Composition

Si p et q sont deux termes, on note $p \circ q$ le terme $\lambda x.q(p\ x)$ (et pas $\lambda x.p(q\ x)$). Donc

$$\begin{array}{rcl} \lambda x.\lfloor \mathbf{d} \rfloor\ ((p \circ q)\ x) & =_\beta & \lambda x.\lfloor \mathbf{d} \rfloor\ (q\ (p\ x)) \\ & =_\beta & \lambda x.(\lambda y.\lfloor \mathbf{d} \rfloor\ (q\ y))\ (p\ x) \end{array}$$

Donc, si $[\![q]\!]\,(\mathbf{d})$ est défini, $[\![p \circ q]\!]\,(\mathbf{d}) = [\![p]\!]\,([\![q]\!]\,(\mathbf{d}))$ On peut écrire le programme calculant la composition

$$\mathsf{compo} \stackrel{\text{def}}{=} \lambda x.\lambda P.\,\mathsf{let}\ \langle p \rangle = P\ \mathsf{in}\ \lambda Q.\,\mathsf{let}\ \langle q \rangle = Q\ \mathsf{in}\ x\ \langle p \circ q \rangle$$

car

$$\begin{array}{rcl} \lambda x.\lfloor [p, q, d_1, \cdots, d_n] \rfloor (\mathsf{compo}\ x) & =_\beta & \lambda x.\mathsf{compo}\ x\ \langle p \rangle\ \langle d \rangle\ \langle d_1 \rangle\ \cdots\ \langle d_n \rangle \\ & =_\beta & \lambda x.x\ \langle p \circ q \rangle\ \langle d_1 \rangle\ \cdots\ \langle d_n \rangle \\ & =_\beta & \lfloor [p \circ q, d_1, \cdots, d_n] \rfloor \end{array}$$

7.4.3 Machine universelle

On peut donner une machine universelle à Λ_R. On pose

$$\mathsf{univ} \stackrel{\text{def}}{=} \lambda x.\lambda p.(\mathsf{run}\,p)\ x$$

Ce programme vérifie

$$\begin{array}{rcl} \lambda x.\lfloor [p, d_1, \cdots, d_n] \rfloor\ (\mathsf{univ}\ x) & =_\beta & \lambda x.\lfloor [p, d_1, \cdots, d_n] \rfloor\ (\lambda p.(\mathsf{run}\,p)\ x) \\ & =_\beta & \lambda x.(\mathsf{run}\,\langle p \rangle)\ x\ \langle d_1 \rangle\ \cdots\ \langle d_n \rangle \\ & =_\beta & \lambda x.p\ x\ \langle d_1 \rangle\ \cdots\ \langle d_n \rangle \\ & =_\beta & \lambda x.\lfloor [d_1, \cdots, d_n] \rfloor\ (p\ x) \end{array}$$

Donc $[\![\mathsf{univ}]\!]\,[p, d_1, \cdots, d_n] = [\![p]\!]\,[d_1, \cdots, d_n]$

7.4.4 Spécialisation

Si p et d sont un programme et une donnée, on peut donner la spécialisation de p à d de la manière suivante

$$\mathbf{spec}(p,d) \stackrel{\text{def}}{=} \lambda x.p\ x\ \langle d\rangle$$

car

$$\begin{array}{rcl}\lambda x.\lfloor[d_1,\cdots,d_n]\rfloor(\mathbf{spec}(p,d)\ x) & =_\beta & \lambda x.\,\mathbf{spec}(p,d)\ x\ \langle d_1\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & \lambda x.p\ x\ \langle d\rangle\ \langle d_1\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & \lambda x.\lfloor[d,d_1,\cdots,d_n]\rfloor\ (p\ x)\end{array}$$

Donc $[\![\mathbf{spec}(p,d)]\!]\,[d_1,\cdots,d_n] = [\![p]\!]\,[d,d_1,\cdots,d_n]$. On peut aussi écrire le spécialiseur

$$\mathsf{spec} \stackrel{\text{def}}{=} \lambda x.\lambda P.\,\mathsf{let}\,\langle p\rangle = P\ \mathsf{in}\ \lambda D.\,\mathsf{let}\,\langle d\rangle = D\ \mathsf{in}\ x\ \langle \underbrace{\lambda x.p\ x\ \langle d\rangle}_{\mathbf{spec}(p,d)}\rangle$$

car

$$\begin{array}{rcl}\lambda x.\lfloor[p,d,d_1,\cdots,d_n]\rfloor(\mathsf{spec}\ x) & =_\beta & \lambda x.\mathsf{spec}\ x\ \langle p\rangle\ \langle d\rangle\ \langle d_1\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & \lambda x.x\ \langle\,\mathbf{spec}(p,d)\rangle\ \langle d_1\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & \lfloor[\mathbf{spec}(p,d),d_1,\cdots,d_n]\rfloor\end{array}$$

7.4.5 Théorème de récursion

On rappelle le Théorème du point fixe de Kleene de la Section 1.4.2. Pour tout programme p, il existe une donnée e qui encode un programme q telle que, pour toute donnée d, le résultat du calcul de p sur l'entrée e d est le même que le résultat du calcul de q sur l'entrée d.

7.4.5.1 Point fixe réflexif

Définition 7.4.5. Un terme Z est un *combinateur de point fixe réflexif* lorsque pour tout terme $t \in \Lambda_R$

$$\mathsf{Z}\,\langle t\rangle =_\beta t\ \langle\,\mathsf{Z}\,\langle t\rangle\rangle$$

L'opérateur Z est exprimable dans Λ_R : si on pose $\omega \stackrel{\text{def}}{=} \lambda f.\lambda X.\,\mathsf{let}\,\langle x\rangle = X\ \mathsf{in}\ f\ \langle x\ \langle x\rangle\rangle$, alors on peut noter

$$\mathsf{Z} \stackrel{\text{def}}{=} \lambda F.\,\mathsf{let}\,\langle f\rangle = F\ \mathsf{in}\ (\omega\ f)\ \langle\omega\ f\rangle$$

Alors

$$\begin{array}{rcl}\mathsf{Z}\,\langle t\rangle & =_\beta & (\omega\ t)\ \langle\omega\ t\rangle \\ & =_\beta & t\ \langle(\omega\ t)\ \langle\omega\ t\rangle\rangle \\ & =_\beta & t\ \langle\,\mathsf{Z}\,\langle t\rangle\rangle\end{array}$$

7.4.5.2 Autre point fixe

Ce point fixe Z permet d'encoder le combinateur de point fixe habituel Y tel que

$$\mathsf{Y}\,t =_\beta t\ (\mathsf{Y}\,t)$$

Posons

$$y \stackrel{\text{def}}{=} \lambda q.\lambda p.p\ ((\mathsf{run}\ \ q)\ p)$$

On a $\vdash y : !((A \to A) \to A) \to ((A \to A) \to A)$. Enfin, posons $\mathsf{Y} \stackrel{\text{def}}{=} \mathsf{Z}\,\langle y\rangle$. On a alors

$$\begin{array}{rcl} \mathsf{Y}\,p & =_\beta & y\ \langle\,\mathsf{Z}\,\langle y\rangle\rangle\ p \\ & =_\beta & p\ ((\mathsf{run}\ \ \langle\,\mathsf{Z}\,\langle y\rangle\rangle)\ p) \\ & =_\beta & p\ ((\mathsf{Z}\,\langle y\rangle)\ p) \\ & =_\beta & p\ (\mathsf{Y}\,p) \end{array}$$

7.4.5.3 Théorème de récursion de Kleene

Soit $p \in \mathcal{P}$. On cherche à construire un programme e tel que

$$\forall \mathbf{d} \in \mathcal{D}^* \quad [\![p]\!]\,(e.\mathbf{d}) = [\![e]\!]\,(\mathbf{d})$$

Posons

$$e \stackrel{\text{def}}{=} \mathsf{Z}\,\langle \lambda q.\lambda x.p\ x\ q\rangle$$

On a en effet, si $\mathbf{d} = [d_1, \cdots, d_n]$

$$\begin{array}{rcl} \lambda x.\lfloor \mathbf{d}\rfloor\ (e\ x) & =_\beta & \lambda x.e\ x\ \langle d_1\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & \lambda x.p\ x\ \langle e\rangle\ \langle d_1\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & \lambda x.\lfloor e.\mathbf{d}\rfloor\ (p\ x) \end{array}$$

On a donc le résultat voulu.

7.4.5.4 Équivalence entre Z et le théorème de récursion

On a montré (Section 7.4.5.3) que l'on pouvait utiliser un combinateur de point fixe récursif Z pour exhiber une solution au théorème de récursion de Kleene. En fait, on peut construire un combinateur Z à partir d'une solution de Kleene.

On sait que pour tout programme p, il existe une solution e satisfaisant l'équation de Kleene. Posons $F : \mathcal{P} \to \mathcal{P}$ une fonction produisant une une solution e pour tout programme p

$$\forall p \in \mathcal{P}, \mathbf{d} \in \mathcal{D}^* \quad [\![F(p)]\!]\,(\mathbf{d}) = [\![p]\!]\,(F(p).\mathbf{d})$$

En particulier, $[\![F(p)]\!]\,(\epsilon) = [\![p]\!]\,[F(p)]$. Donc, pour tout terme t, $F(p)\ t =_\beta p\ t\ \langle F(p)\rangle$. Posons $\mathsf{Z} \stackrel{\text{def}}{=} F(\lambda\langle p\rangle.\lambda\langle q\rangle.p\ \langle q\ \langle p\rangle\rangle)$. En effet, on a

$$\begin{array}{rcl} \mathsf{Z}\,\langle p\rangle & =_\beta & F(\lambda\langle p\rangle.\lambda\langle q\rangle.p\ \langle q\ \langle p\rangle\rangle)\ \langle p\rangle \\ & =_\beta & p\ \langle\,\mathsf{Z}\,\langle p\rangle\rangle \end{array}$$

7.4.5.5 Interpréteur récursif

On cherche à exprimer un interpréteur récursif $\mathsf{int}\omega$ dans Λ_R. Posons $\delta \stackrel{\text{def}}{=} \lambda e.\lambda x.\lambda D.\,\mathsf{let}\,\langle d\rangle = D\ \mathsf{in}\ d\ x\ e$ et $h \stackrel{\text{def}}{=} \mathsf{Z}\,\langle\delta\rangle$. Le terme h est un interpréteur récursif car

$$\begin{array}{rcl} [\![h]\!]\,[d_1, \cdots, d_n] & =_\beta & \lambda x.\lfloor [d_1, \cdots, d_n]\rfloor\ (h\ x) \\ & =_\beta & \lambda x.\lfloor [d_1, \cdots, d_n]\rfloor\ (\delta\ \langle h\rangle\ x) \\ & =_\beta & \lambda x.(\delta\ \langle h\rangle\ x)\langle d_1\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & \lambda x.d_1\ x\ \langle h\rangle\ \langle d_2\rangle\ \cdots\ \langle d_n\rangle \\ & =_\beta & [\![d_1]\!]\,[h, d_2, \cdots, d_n] \end{array}$$

7.5 Staged computation

Notre calcul exprime un paradigme connu depuis [JS86], le *calcul par niveaux*, ou *staged computation*. L'idée est de séparer les temps d'exécution des termes. La manière la plus simple est de séparer les termes à exécuter maintenant (compile-time) des termes à exécuter plus tard (run-time) [GJ91 ; JGS93]. L'analyse des liaisons des variables (*binding time analysis*) (comme dans [NN92]) montre qu'il est possible d'établir un système de types permettant de spécifier l'état d'évaluation d'un terme (compile-time ou run-time)

Cette section a pour but de rappeler deux grandes méthodes présentes dans la littérature, que l'on souhaite mettre en avant car elle se fondent avant tout sur une vue *logique* et non *algorithmique*.

On se borne ici a transposer les notations des auteurs dans les nôtres et on réexprime la définition des langages dans notre cadre d'étude (langages confluents sans stratégie). On prouve que ces langages sont confluents dans leur nouvelle description.

La première permet de définir des termes dont l'exécution est retardée. La seconde permet de répartir le calcul sur plusieurs niveaux.

7.5.1 Première approche : retardement de termes

Cette méthode s'attache à retarder la réduction de termes, plutôt que forcer la réduction de certains sous-termes. Cette méthode est due en partie à Davis et Pfenning dans [DP96]. Ici, on s'intéresse moins aux capacités du langage pour l'évaluation partielle qu'à l'étude d'un langage dans lequel une partie des termes est retardée. Cette méthode est *en tout point* semblable à la nôtre, bien qu'établie dans un cadre logique différent. Nous verrons dans la Section 9.6.2 qu'il existe un lien entre le travail en logique modale et le nôtre.

La sémantique de ce langage Λ_M est exactement la même que notre langage, seule sa syntaxe change. C'est pourquoi, on va juste faire un rappel de sa syntaxe et montrer la correspondance entre Λ_M et Λ_R.

7.5.1.1 Syntaxe

La syntaxe de Λ_M est donnée par

$$\Lambda_M \ni t \stackrel{\text{def}}{=} x \mid \lambda x.t \mid t\ t \mid \mathsf{box}(t) \mid \mathsf{let\,box}(t) = t\ \mathsf{in}\ t$$

On passe d'un terme de Λ_M à un terme de Λ_R en remplaçant tous les $\mathsf{box}(t)$ par $\langle t \rangle$.

7.5.2 Deuxième approche : nivellement du calcul

Cette méthode permet de forcer la réduction de sous-termes. Cette façon de faire est théorisée dans les travaux de Davies [Dav96] , ou encore dans ceux de Taha et Sheard [TS97]. On va adapter le travail de ces auteurs au nôtre, notamment en donnant une version non typée et sans stratégie de réduction (on pourra ainsi parler à nouveau de confluence par exemple). Cette approche se base sur la logique temporelle, comme on le verra dans la Section 9.6.1.

7.5.2.1 Syntaxe

En reprenant les notations de Taha et Sheard, la syntaxe du λ-calcul est étendue avec les *termes retardés* $\langle t \rangle$ [3] et *échappés* $\sim t$.

$$\Lambda_T \ni t \stackrel{\text{def}}{=} x \mid t\ t \mid \lambda x.t \mid \langle t \rangle \mid \sim t$$

7.5.2.2 Contextes de structure

Les contextes de structure sont définis de la même manière qu'en λ-calcul

$$S \stackrel{\text{def}}{=} [] \mid S\ t \mid t\ S \mid \lambda x.S \mid \langle S \rangle \mid \sim S$$

7.5.2.3 Niveaux

Le *niveau* $\Delta(S)$ d'un contexte de structure est défini par induction de la manière suivante.

$$\begin{aligned}
\Delta([]) &\stackrel{\text{def}}{=} 0 \\
\Delta(S\ t) &\stackrel{\text{def}}{=} \Delta(S) \\
\Delta(t\ S) &\stackrel{\text{def}}{=} \Delta(S) \\
\Delta(\lambda x.S) &\stackrel{\text{def}}{=} \Delta(S) \\
\Delta(\langle S \rangle) &\stackrel{\text{def}}{=} \Delta(S) + 1 \\
\Delta(\sim S) &\stackrel{\text{def}}{=} \Delta(S) - 1
\end{aligned}$$

Remarque 7.5.1. Le niveau d'un contexte de structure peut être négatif. Par exemple $\Delta(\sim []) = -1$. ■

Le niveau d'un sous-terme t' de t est le niveau du contexte de structure S tel que $t = S[t']$. C'est le degré d'imbrication des $\langle \cdots \rangle$ (positif) et des $\sim$ (négatif).

7.5.2.4 Contextes de réduction

Les contextes de réduction sont définis de la même manière qu'en λ-calcul, sauf qu'on y ajoute une notion de niveau, c'est à dire qu'ils sont indexés par les entiers naturels n.

De plus, comme on le verra, deux redexes sont possibles, et ils sont contractables dans des contextes de réduction différents : on définit donc deux contextes de réduction E^{λ}_n et $E^{\sim}_n$.

$$\begin{aligned}
E^{\lambda}_0 &= [] \mid \lambda x.E^{\lambda}_0 \mid E^{\lambda}_0\ t \mid t\ E^{\lambda}_0 \mid \langle E^{\lambda}_1 \rangle \\
E^{\lambda}_{n+1} &= \lambda x.E^{\lambda}_{n+1} \mid E^{\lambda}_{n+1}\ t \mid t\ E^{\lambda}_{n+1} \mid \langle E^{\lambda}_{n+2} \rangle \mid \sim E^{\lambda}_n \\
E^{\sim}_0 &= \lambda x.E^{\sim}_0 \mid E^{\sim}_0\ t \mid t\ E^{\sim}_0 \mid \langle E^{\sim}_1 \rangle \\
E^{\sim}_1 &= [] \mid \lambda x.E^{\sim}_1 \mid E^{\sim}_1\ t \mid t\ E^{\sim}_1 \mid \langle E^{\sim}_2 \rangle \mid \sim E^{\sim}_0 \\
E^{\sim}_{n+2} &= \lambda x.E^{\sim}_{n+2} \mid E^{\sim}_{n+2}\ t \mid t\ E^{\sim}_{n+2} \mid \langle E^{\sim}_{n+2} \rangle \mid \sim E^{\sim}_{n+1}
\end{aligned}$$

Propriété 7.5.2. *Pour tout contexte de structure de la forme E^{λ}_n (resp. $E^{\sim}_n$), $\Delta(E^{\lambda}_n) = -n$ (resp. $\Delta(E^{\sim}_n) = 1 - n$).*

Démonstration. Par induction sur le contexte de structure. □

3. On prend volontairement la même notation que dans Λ_R, car les termes $\langle t \rangle$ de Λ_T et Λ_R sont retardés dans les deux cas

7.5.2.5 Liaison des variables et α-renommage

L'ensemble des variables libres $\mathrm{FV}(t)$ d'un terme t est l'union des ensembles de variables libres $\mathrm{FV}_\delta(t)$ des sous-termes de niveau δ de t.

$$\begin{array}{rcll}
\mathrm{FV}_0(x) & \stackrel{\text{def}}{=} & \{x\} & \\
\mathrm{FV}_\delta(x) & \stackrel{\text{def}}{=} & \emptyset & \delta \neq 0 \\
\mathrm{FV}_\delta(t_1\ t_2) & \stackrel{\text{def}}{=} & \mathrm{FV}_\delta(t_1) \cup \mathrm{FV}_\delta(t_2) & \\
\mathrm{FV}_0(\lambda x.t) & \stackrel{\text{def}}{=} & \mathrm{FV}_0(t) \setminus \{x\} & \\
\mathrm{FV}_\delta(\lambda x.t) & \stackrel{\text{def}}{=} & \mathrm{FV}_\delta(t) & \delta \neq 0 \\
\mathrm{FV}_\delta(\langle t \rangle) & \stackrel{\text{def}}{=} & \mathrm{FV}_{\delta-1}(t) & \\
\mathrm{FV}_\delta(\sim t) & \stackrel{\text{def}}{=} & \mathrm{FV}_{\delta+1}(t) &
\end{array}$$

On pose

$$\mathrm{FV}(t) \stackrel{\text{def}}{=} \bigcup_{\delta \in \mathbb{Z}} \mathrm{FV}_\delta(t)$$

Exemple 7.5.3. On a les résultats suivants

1. $\mathrm{FV}_1(\lambda x.\langle x \rangle) = \{x\}$
2. $\mathrm{FV}_0(\langle x \rangle) = \emptyset$ et $\mathrm{FV}_1(\langle x \rangle) = \{x\}$
3. $\mathrm{FV}_0(x\ \langle x \rangle) = \mathrm{FV}_1(x\ \langle x \rangle) = \{x\}$

L'Exemple 1 montre qu'un lieur λ ne peut lier que des variables libres de même niveau que lui : il est de niveau 0 et x est de niveau 1, donc pas de liaison possible. L'Exemple 2 lie le niveau d'une variable avec sa définition de liberté. Il est généralisé par la Propriété 7.5.4. L'Exemple 3 donne un cas où un même nom de variable est utilisé pour deux choses différentes : le point 1 a montré qu'un lieur ne pouvait être lié qu'à une variable de même niveau que lui. Donc les deux occurrences de x dans $x\ \langle x \rangle$ ne pourront être liées par le même lieur. Ce terme est équivalent à $x\ \langle y \rangle$. ■

Propriété 7.5.4. *Soit* $x \in \mathrm{FV}(t)$.

$$x \in \mathrm{FV}_\delta(t) \iff \exists S, S[x] = t \wedge \Delta(S) = \delta$$

Démonstration. Par induction sur t. □

On considère à présent les termes modulo α-renommage, afin d'éviter le point 3.

7.5.2.6 Substitution

Les règles de substitution sont les mêmes qu'en λ-calcul

$$\begin{array}{rcll}
x[x/t'] & \stackrel{\text{def}}{=} & t' & \\
y[x/t'] & \stackrel{\text{def}}{=} & y & x \neq y \\
(\lambda y.t)[x/t'] & \stackrel{\text{def}}{=} & \lambda y.t[x/t'] & x \neq y \\
(t_1\ t_2)[x/t'] & \stackrel{\text{def}}{=} & t_1[x/t']\ t_2[x/t'] & \\
\langle t \rangle[x/t'] & \stackrel{\text{def}}{=} & \langle t[x/t'] \rangle & \\
(\sim t)[x/t'] & \stackrel{\text{def}}{=} & \sim t[x/t'] &
\end{array}$$

Propriété 7.5.5. *Si* $x \notin \mathrm{FV}(t)$, *alors* $t[x/t'] = t$.

Démonstration. Par induction sur t. □

7.5.2.7 Réduction

Par rapport au λ-calcul, on ajoute un nouveau redex

$$\begin{array}{rcl} E_n^\lambda[(\lambda x.t)\ t'] & \underset{n}{\overset{\lambda}{\to}} & E_n^\lambda[t[x/t']] \\ E_n^\sim[\sim\langle t\rangle] & \underset{n}{\overset{\sim}{\to}} & E_n^\sim[t] \end{array}$$

On pose $\overset{n}{\to} \overset{\text{def}}{=} \underset{n}{\overset{\lambda}{\to}} \cup \underset{n}{\overset{\sim}{\to}}$.

Remarque 7.5.6. On notera qu'il n'existe pas de contexte E_0^λ de la forme $\sim S$. Pour des raisons de typage que l'on verra plus loin, on ne peut pas réduire au niveau 0 derrière un $\sim$. On notera aussi qu'il n'y a pas de "trou" dans les contextes de réduction E_{n+1}^λ. On ne réduit donc pas les redexes $(\lambda x.t)\ t'$ des niveaux non nuls. Par exemple, les termes $(\lambda x.x)\ (\lambda y.y)$ et $\langle \sim ((\lambda x.x)\ (\lambda y.y))\rangle$ se réduisent à $\lambda y.y$ et $\langle \sim (\lambda y.y)\rangle$ mais $\langle(\lambda x.x)\ (\lambda y.y)\rangle$ ne se réduit pas.

De même, on ne réduit pas de redexes $\sim\langle t\rangle$ de niveaux différents de 1. Par exemple, le terme $\langle \sim \langle\lambda x.x\rangle\rangle$ se réduit à $\langle\lambda x.x\rangle$, mais les termes $\langle\langle \sim \langle\lambda x.x\rangle\rangle\rangle$ et $\sim\langle\lambda x.x\rangle$ ne se réduisent pas. ■

7.5.2.8 Confluence

Théorème 7.5.7. *Les systèmes* $(\Lambda_T, \overset{n}{\to})$ *sont confluents.*

On prouve ce résultat en se ramenant à la confluence du λ-calcul avec les constantes $c_{var}, c_\lambda, c_{app}, c_{box}, c_\sim, c_{err}, d_{box}$ et le nouveau redex $d_{box}\ (c_{box}\ t) \overset{\delta}{\to} t$, c'est à dire que si E est un contexte de réduction du λ-calcul (voir Section 5.1.6), on a

$$\begin{array}{rcl} E[(\lambda x.t)\ t'] & \overset{\lambda}{\to} & E[t[x/t']] \\ E[d_{box}\ (c_{box}\ t)] & \overset{\delta}{\to} & E[t] \end{array}$$

Le système $(\Lambda, \overset{\lambda}{\to} \cup \overset{\delta}{\to})$ est confluent, comme le montre la Propriété 7.5.11.

Pour $n \in \mathbb{N}$, on introduit alors une transformation $\mathcal{F}_n : \Lambda_T \to \Lambda$ vérifiant les propriétés suivantes.

Propriété 7.5.8. *Si* $t \overset{n}{\to} t'$, *alors* $\mathcal{F}_n(t) \overset{*}{\to} \mathcal{F}_n(t')$

Propriété 7.5.9. *Si* $\mathcal{F}_n(t) \to T$, *alors il existe* $t' \in \Lambda_T$ *tel que* $\mathcal{F}_n(t') = T$ *et* $t \overset{n}{\to} t'$.

Par ces deux propriétés, Λ_T hérite de la confluence de Λ. Posons

$$\begin{array}{rcl} \mathcal{F}_0(x) & \overset{\text{def}}{=} & x \\ \mathcal{F}_{n+1}(x) & \overset{\text{def}}{=} & c_{var} \\ \mathcal{F}_0(t\ t') & \overset{\text{def}}{=} & \mathcal{F}_0(t)\ \mathcal{F}_0(t') \\ \mathcal{F}_{n+1}(t\ t') & \overset{\text{def}}{=} & c_{app}\ \mathcal{F}_{n+1}(t)\ \mathcal{F}_{n+1}(t') \\ \mathcal{F}_0(\lambda x.t) & \overset{\text{def}}{=} & \lambda x.\mathcal{F}_0(t) \\ \mathcal{F}_{n+1}(\lambda x.t) & \overset{\text{def}}{=} & c_\lambda\ \mathcal{F}_{n+1}(t) \\ \mathcal{F}_n(\langle t\rangle) & \overset{\text{def}}{=} & c_{box}\ \mathcal{F}_{n+1}(t) \\ \mathcal{F}_0(\sim t) & \overset{\text{def}}{=} & c_{err} \\ \mathcal{F}_1(\sim t) & \overset{\text{def}}{=} & d_{box}\ \mathcal{F}_0(t) \\ \mathcal{F}_{n+2}(\sim t) & \overset{\text{def}}{=} & c_\sim\ \mathcal{F}_{n+1}(t) \end{array}$$

La Propriété 7.5.8 se montre par cas selon le contexte de réduction (E^{λ}_n ou $E^{\sim}_n$) utilisé pour $\overset{n}{\to}$, puis par induction sur le contexte. On utilise pour cela le lemme suivant

Lemme 7.5.10. *Si* $\forall \delta \neq -n, x \notin \mathrm{FV}_\delta(t)$ *alors* $\mathcal{F}_n(t[x/t']) = \mathcal{F}_n(t)[x/\mathcal{F}_0(t')]$

Démonstration. Par induction sur t.

— Si $t = x$, alors $x \in \mathrm{FV}_0(t)$. D'autre part $t[x/t'] = t'$ et $\mathcal{F}_0(t) = x$.

— Si $\forall \delta \neq -n, x \notin \mathrm{FV}_\delta(\langle t\rangle) = \mathrm{FV}_{\delta-1}(t)$, alors $\mathcal{F}_n(\langle t\rangle[x/t']) = c_{box}\ \mathcal{F}_{n+1}(t[x/t']) = c_{box}\ \mathcal{F}_{n+1}(t)[x/\mathcal{F}_0(t')] = \mathcal{F}_n(\langle t\rangle)[x/\mathcal{F}_0(t')]$.

— Si $\forall \delta \neq -(n+2), x \notin \mathrm{FV}_\delta(\sim t) = \mathrm{FV}_{\delta+1}(t)$, alors $\mathcal{F}_{n+2}((\sim t)[x/t']) = c_\sim\ \mathcal{F}_{n+1}(t[x/t']) = c_\sim \mathcal{F}_{n+1}(t)[x/\mathcal{F}_0(t')] = \mathcal{F}_n(\sim t)[x/\mathcal{F}_0(t')]$.

— $\mathcal{F}_0((\sim t)[x/t']) = c_{err} = \mathcal{F}_0(\sim t)[x/\mathcal{F}_0(t')]$.

— $\mathcal{F}_1((\sim t)[x/t']) = d_{box}\ \mathcal{F}_0(t[x/t']) = d_{box}\ \mathcal{F}_0(t)[x/\mathcal{F}_0(t')] = \mathcal{F}_1(\sim t)[x/t']$.

— On a $\mathcal{F}_0((t_1\ t_2)[x/t']) = \mathcal{F}_0(t_1[x/t'])\ \mathcal{F}_0(t_2[x/t'])$. Or on sait que $\forall \delta \neq 0, x \notin \mathrm{FV}_0(t_1\ t_2) = \mathrm{FV}_0(t_1) \cup \mathrm{FV}_0(t_2)$. Donc $x \notin \mathrm{FV}_0(t_1)$ et $x \in \mathrm{FV}_0(t_2)$. Par induction, $\mathcal{F}_0((t_1\ t_2)[x/t']) = \mathcal{F}_0(t_1)[x/\mathcal{F}_0(t')]\ \mathcal{F}_0(t_2)[x/\mathcal{F}_0(t')] = \mathcal{F}_0(t_1\ t_2)[x/\mathcal{F}_0(t')]$. De même pour $\mathcal{F}_{n+1}((t_1\ t_2)[x/t'])$.

— Si $\forall \delta \neq 0, x \notin \mathrm{FV}_\delta(\lambda y.t) = \mathrm{FV}_0(t)$, alors $\mathcal{F}_0((\lambda y.t)[x/t']) = \lambda y.\mathcal{F}_0(t[x/t']) = \lambda y.\mathcal{F}_0(t)[x/\mathcal{F}_0(t')] = \mathcal{F}_0(\lambda y.t)[x/\mathcal{F}_0(t')]$.

— Si $\forall \delta \neq -(n+1), x \notin \mathrm{FV}_\delta(\lambda y.t)$, alors $\forall \delta \notin \{-(n+1), 0\}, x \notin \mathrm{FV}_\delta(t)$. De plus $x \notin \mathrm{FV}_0(t)$ ou $x = y$, or on peut renommer y et $y' \neq x$. Donc $x \notin \mathrm{FV}_0(t)$. Donc $\mathcal{F}_{n+1}((\lambda y.t)[x/t']) = c_\lambda\ \mathcal{F}_{n+1}(t[x/t']) = c_\lambda \mathcal{F}_{n+1}(t)[x/\mathcal{F}_0(t')] = \mathcal{F}_{n+1}(\lambda y.t)[x/\mathcal{F}_0(t')]$.

□

La preuve de la Propriété 7.5.8 se fait par induction sur les contextes. Par exemple,

— Si $(\lambda x.t)\ t' \overset{0}{\to} t[x/t']$, on sait que $x \notin \mathrm{FV}_\delta(t)$, pour tout $\delta \neq 0$. Donc, par le Lemme 7.5.10, $\mathcal{F}_0(t[x/t']) = \mathcal{F}_0(t)[x/\mathcal{F}_0(t')]$.
Donc $\mathcal{F}_0((\lambda x.t)\ t') = \mathcal{F}_0(\lambda x.t)\ \mathcal{F}_0(t') = (\lambda x.\mathcal{F}_0(t))\ \mathcal{F}_0(t') \to \mathcal{F}_0(t)[x/\mathcal{F}_0(t')] = \mathcal{F}_0(t[x/t'])$

— $\sim \langle t\rangle \overset{1}{\to} t$ et $\mathcal{F}_1(\sim \langle t\rangle) = d_{box}\ (c_{box}\ \mathcal{F}_1(t)) \to \mathcal{F}_1(t)$.

— Si $\langle t\rangle \overset{n}{\to} \langle t'\rangle$. Ce cas sera traité dans les détails, les autres le seront plus rapidement. On sait que $\langle t\rangle = \langle E^{\lambda}_{n+1}((\lambda x.t_1)\ t_2)\rangle$ et que $\langle t'\rangle = \langle E^{\lambda}_{n+1}(t_1[x/t_2])\rangle$, ou alors que $\langle t\rangle = \langle E^{\sim}_{n+1}(\sim \langle t_1\rangle)$ et que $\langle t'\rangle = \langle E^{\sim}_{n+1}(t_1)\rangle$. Donc, comme $\langle t\rangle \overset{n}{\to} \langle t'\rangle$, alors $E^{\lambda}_{n+1}((\lambda x.t_1)\ t_2) \overset{n+1}{\to} E^{\lambda}_{n+1}(t_1[x/t_2])$ ou $E^{\sim}_{n+1}(\sim \langle t_1\rangle) \overset{n+1}{\to} E^{\sim}_{n+1}(t_1)$. Dans les deux cas $t \overset{n+1}{\to} t'$. Donc $c_{box}\ \mathcal{F}_{n+1}(t) \to c_{box}\ \mathcal{F}_{n+1}(t')$ et finalement $\mathcal{F}_n(\langle t\rangle) \to \mathcal{F}_n(\langle t'\rangle)$

— Si $\sim t \overset{n}{\to} \sim t'$ alors $n \neq 0$ et $t \overset{n-1}{\to} t'$. Donc $\mathcal{F}_{n-1}(t) \to \mathcal{F}_{n-1}(t')$. Si $n = 1$, $d_{box}\ \mathcal{F}_0(t) \to d_{box}\ \mathcal{F}_0(t)$ et donc $\mathcal{F}_1(\sim t) \to \mathcal{F}_1(\sim t')$. Si $n \neq 1$, alors $c_\sim\ \mathcal{F}_{n-1}(t) \to c_\sim\ \mathcal{F}_{n-1}(t')$ et donc $\mathcal{F}_n(\sim t) \to \mathcal{F}_n(\sim t')$.

— Si $\lambda x.t \overset{n}{\to} \lambda x.t'$, alors $t \overset{n}{\to} t'$. Donc $\mathcal{F}_n(t) \to \mathcal{F}_n(t')$. Si $n \neq 0$, on a directement que $\mathcal{F}_n(\lambda x.t) \to \mathcal{F}_n(\lambda x.t')$. Sinon, on a que $\lambda x.\mathcal{F}_0(t) \to \lambda x.\mathcal{F}_0(t')$, et donc que $\mathcal{F}_0(\lambda x.t) \to \mathcal{F}_0(\lambda x.t')$.

— Si $t_1\ t_2 \xrightarrow{n} t'_1\ t_2$, alors $t_1 \xrightarrow{n} t'_1$. Donc $\mathcal{F}_n(t_1) \to \mathcal{F}_n(t'_1)$. Si $n \neq 0$, alors $1\ \mathcal{F}_n(t_1)\ \mathcal{F}_n(t_2) \to 1\ \mathcal{F}_n(t'_1)\ \mathcal{F}_n(t_2)$ et donc $\mathcal{F}_n(t_1\ t_2) \to \mathcal{F}_n(t'_1\ t_2)$. De plus, $\mathcal{F}_0(t_1)\ \mathcal{F}_0(t_2) \to \mathcal{F}_0(t'_1)\ \mathcal{F}_0(t_2)$ donc $\mathcal{F}_0(t_1\ t_2) \to \mathcal{F}_0(t'_1\ t_2)$.

La preuve de la Propriété 7.5.9 se fait par induction sur le contexte de réduction de $\mathcal{F}_n(t)$.

— Si $\mathcal{F}_n(t)$ se réduit "à son sommet" par $\xrightarrow{\lambda}$, alors, $n = 0$ et $t = (\lambda x.t_1)\ t_2$ et $\mathcal{F}_0(t) = \lambda x.\mathcal{F}_0(t_1)\ \mathcal{F}_0(t_2)$ qui se réduit en $\mathcal{F}_0(t_1)[x/\mathcal{F}_0(t_2)] = \mathcal{F}_0(t_1[x/t_2])$ (Lemme 7.5.10).

— Si $\mathcal{F}_n(t)$ se réduit "à son sommet" par la règle de destruction, alors, $n = 1$ et $t =\sim\ \langle t_1\rangle$ et $\mathcal{F}_1(t) = d_{box}\ (c_{box}\ \mathcal{F}_1(t_1))$ qui se réduit en $\mathcal{F}_1(t_1)$.

— Les autres cas sont de simples appels à l'hypothèse d'induction.

Propriété 7.5.11. *Le système* $(\Lambda, \xrightarrow{\lambda} \cup \xrightarrow{\delta})$ *est confluent.*

Démonstration. La preuve de cette confluence utilise à nouveau le lemme de Hindley-Rosen [Ros73 ; Hin64] : il suffit de montrer que $(\Lambda, \xrightarrow{\lambda})$ et $(\Lambda, \xrightarrow{\delta})$ sont confluents, et que ces deux systèmes commutent.

— La confluence de $(\Lambda, \xrightarrow{\lambda})$ est abordée dans la Section 5.1.8.

— Le système $(\Lambda, \xrightarrow{\delta})$ est fortement confluent : si R et S sont deux redexes et R est un redex $d_{box}\ (c_{box}\ t)$, alors, soit S est un sous-terme de t, et dans ce cas la réduction de R ne change pas S, soit S n'est pas dans t, et dans ce cas non plus S n'est pas changé. Si S est lui aussi un δ-redex, on peut donc réduire S et R de manière indépendante.

— Les systèmes $(\Lambda, \xrightarrow{\lambda})$ et $(\Lambda, \xrightarrow{\delta})$ commutent. En effet, soient R et S un λ-redex et un δ-redex. La contraction de S ne duplique pas de R, comme on l'a vu dans le point précédent. D'autre part, la contraction de R peut créer différentes copies de S. Donc, pour tout t, t_1, t_2 tel que $t \xrightarrow{\lambda} t_1$ et $t \xrightarrow{\delta} t_2$, alors il existe s tel que $t_1 \xrightarrow{*\delta} s$ et $t_2 \xrightarrow{\lambda} s$.

On en déduit que si $t \xrightarrow{\lambda} t_1$ et $t \xrightarrow{*\delta} t_2$, alors il existe s tel que $t_2 \xrightarrow{\lambda} s$ et $t_1 \xrightarrow{*\delta} s$ (par induction sur la longueur de $t \xrightarrow{*\delta} t_2$).

On en déduit que si $t \xrightarrow{*\lambda} t_1$ et $t \xrightarrow{*\delta} t_2$, alors il existe s tel que $t_1 \xrightarrow{*\delta} s$ et $t_2 \xrightarrow{*\lambda} s$ (par induction sur la longueur de $t \xrightarrow{*\lambda} t_1$).

□

7.5.2.9 Application : évaluation partielle

On étend la syntaxe du langage Λ_T à Λ_T^{ML} avec un point fixe, les entiers et la conditionnelle if.

$$\Lambda_T^{\mathrm{ML}} \ni t \stackrel{\text{def}}{=} n \mid x \mid \lambda x.t \mid t\ t \mid \langle t\rangle \mid \sim t \mid \mathsf{fix}\, x.t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$$

Les nouveaux redexes sont les suivants

$$\begin{aligned} \mathsf{if}\ 1\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 &\xrightarrow{\mathsf{if}} t_1 \\ \mathsf{if}\ 0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 &\xrightarrow{\mathsf{if}} t_2 \\ \mathsf{fix}\, x.t &\xrightarrow{\mathsf{fix}} t[x/\,\mathsf{fix}\, x.t] \end{aligned}$$

Dans la suite on fixe une stratégie d'évaluation de Λ_T^{ML}. On choisit la stratégie CBV en fixant les contextes d'évaluation Φ et les termes v non évaluables en CBV (on nomme ces termes *valeurs*).

Cet ensemble de valeurs est donné récursivement par

$$\begin{aligned} v_0 &\stackrel{\text{def}}{=} n \mid \lambda x.t \mid \langle v_1 \rangle \\ v_1 &\stackrel{\text{def}}{=} n \mid \lambda x.v_1 \mid x \mid v_1\ v_1 \mid \mathsf{fix}\, x.v_1 \mid \mathsf{if}\ v_1\ \mathsf{then}\ v_1\ \mathsf{else}\ v_1 \mid \langle v_2 \rangle \\ v_{n+2} &\stackrel{\text{def}}{=} n \mid \lambda x.v_{n+2} \mid x \mid v_{n+2}\ v_{n+2} \mid \mathsf{fix}\, x.v_{n+2} \mid \mathsf{if}\ v_{n+2}\ \mathsf{then}\ v_{n+2}\ \mathsf{else}\ v_{n+2} \mid \langle v_{n+2} \rangle \mid \sim v_{n+1} \end{aligned}$$

Les contextes d'évaluations sont également de deux formes Φ_n^λ et $\Phi_n^\sim$

$$\begin{aligned} \Phi_0^\lambda &\stackrel{\text{def}}{=} [] \mid \Phi_0^\lambda\ t \mid v_0\ \Phi_0^\lambda \mid \langle \Phi_1^\lambda \rangle \mid \mathsf{if}\ \Phi_0^\lambda\ \mathsf{then}\ t\ \mathsf{else}\ t \\ \Phi_{n+1}^\lambda &\stackrel{\text{def}}{=} \Phi_{n+1}^\lambda\ t \mid v_{n+1}\ \Phi_{n+1}^\lambda \mid \langle \Phi_{n+2}^\lambda \rangle \mid \sim \Phi_n^\lambda \mid \lambda x.\Phi_{n+1}^\lambda \mid \mathsf{fix}\, x.\Phi_{n+1}^\lambda \\ &\quad \mid \mathsf{if}\ \Phi_{n+1}^\lambda\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{if}\ v_{n+1}\ \mathsf{then}\ \Phi_{n+1}^\lambda\ \mathsf{else}\ t \mid \mathsf{if}\ v_{n+1}\ \mathsf{then}\ v_{n+1}\ \mathsf{else}\ \Phi_{n+1}^\lambda \end{aligned}$$

$$\begin{aligned} \Phi_0^\sim &\stackrel{\text{def}}{=} \Phi_0^\sim\ t \mid v_0\ \Phi_0^\sim \mid \langle \Phi_1^\sim \rangle \mid \mathsf{if}\ \Phi_0^\sim\ \mathsf{then}\ t\ \mathsf{else}\ t \\ \Phi_1^\sim &\stackrel{\text{def}}{=} [] \mid \Phi_1^\sim\ t \mid v_1\ \Phi_1^\sim \mid \langle \Phi_2^\sim \rangle \mid \sim \Phi_0^\sim \mid \lambda x.\Phi_1^\sim \mid \mathsf{fix}\, x.\Phi_1^\sim \\ &\quad \mid \mathsf{if}\ \Phi_1^\sim\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{if}\ v_1\ \mathsf{then}\ \Phi_1^\sim\ \mathsf{else}\ t \mid \mathsf{if}\ v_1\ \mathsf{then}\ v_1\ \mathsf{else}\ \Phi_1^\sim \\ \Phi_{n+2}^\sim &\stackrel{\text{def}}{=} \Phi_{n+2}^\sim\ t \mid v_{n+2}\ \Phi_{n+2}^\sim \mid \langle \Phi_{n+3}^\sim \rangle \mid \sim \Phi_{n+1}^\sim \mid \lambda x.\Phi_{n+2}^\sim \mid \mathsf{fix}\, x.\Phi_{n+2}^\sim \\ &\quad \mid \mathsf{if}\ \Phi_{n+2}^\sim\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{if}\ v_{n+2}\ \mathsf{then}\ \Phi_{n+2}^\sim\ \mathsf{else}\ t \mid \mathsf{if}\ v_{n+2}\ \mathsf{then}\ v_{n+2}\ \mathsf{else}\ \Phi_{n+2}^\sim \end{aligned}$$

Les règles d'évaluations sont données par

$$\begin{aligned} \Phi_n^\lambda[(\lambda x.t)\ t'] &\xrightarrow[n]{\lambda} \Phi_n^\lambda[t[x/t']] \\ \Phi_n^\lambda[\mathsf{fix}\, x.t] &\xrightarrow[n]{\mathsf{fix}} \Phi_n^\lambda[t[x/\,\mathsf{fix}\, x.t]] \\ \Phi_n^\lambda[\mathsf{if}\ 1\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2] &\xrightarrow[n]{\mathsf{if}} \Phi_n^\lambda[t_1] \\ \Phi_n^\lambda[\mathsf{if}\ 0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2] &\xrightarrow[n]{\mathsf{if}} \Phi_n^\lambda[t_2] \\ \Phi_n^\sim[\sim \langle t \rangle] &\xrightarrow[n]{\sim} \Phi_n^\sim[t_1] \end{aligned}$$

Considérons le programme calculant la puissance n-ème d'un entier a. On peut écrire ce programme en λ-calcul avec la fonction suivante :

$$\mathsf{pow} \stackrel{\text{def}}{=} \mathsf{fix}\, p.\lambda n.\mathsf{if}\ n\ \mathsf{then}\ \lambda a.a \times p\ (n-1)\ a\ \mathsf{else}\ \lambda a.1$$

Étant donnée la stratégie de réduction en appel par valeur, $\mathsf{pow}\,2$ se réduit en

$$\mathsf{pow}\,2 \overset{\mathrm{CBV}*}{\to} \lambda a.a \times \mathsf{pow}\ \ 1\ a$$

ce qui n'est pas la forme optimale que l'on attendrait : $\lambda a.a \times a$. Dans cet exemple on aimerait que le terme $a \times \mathsf{pow}\,1\ a \overset{\mathrm{CBN}*}{\to} a \times a$ soit évalué (toujours dans la stratégie de départ, ici en appel par nom) avant le terme global, ou, ce qui revient au même, que l'exécution du terme global soit retardée.

7.5.2.10 Solution possible

Le terme pow est exprimable dans Λ_T^{ML} d'une manière plus optimale

$$\begin{array}{l}\mathsf{pow} \stackrel{\text{def}}{=} \lambda n. \\ \quad \langle \lambda a. \\ \quad\quad \sim (\\ \quad\quad\quad (\mathsf{fix}\, \lambda p.\lambda n'.\mathsf{if}\ n' = 0\ \mathsf{then}\ \langle 1\rangle\ \mathsf{else}\ \langle a \times \sim (p\ (n-1))\rangle)\ n \\ \quad\quad) \\ \quad \rangle\end{array}$$

Par optimal, on l'entend dans le sens de la spécialisation de pow. En effet, pow 2 se réduit à $\langle \lambda a.a \times (a \times 1)\rangle$.

Il existe des langages de programmation où cette notion de retardement est implémentée. Par exemple, en LISP, une expression peut être gelée en utilisant la syntaxe ‘ (équivalent de $\langle\rangle$). Par exemple, l'expression ‘(+ 1 2) se réduit à (+ 1 2) et non à 3. Il est également possible de spécifier qu'une partie du code retardé est à exécuter de manière anticipée, grâce à la syntaxe , (équivalent de $\sim$). Par exemple, le terme ‘(+ 1 ,(+ 1 1)) se réduit à (+ 1 2).

On peut alors définir une fonction pow en LISP qui se réduit de manière optimale (c'est à dire complètement) :

```
(defun pow(n)
  `(lambda a
     ,(((fix (lambda p
        (lambda n' (if (eq n' 0)
           `1
           `(* a ,(p (- n 1)))
        )))) (n))
)))
```

En effet, pow 2 se réduit dans ce cas à `(lambda (a) (* a (* a 1))).

7.5.2.11 Comparaison avec Λ_R

Le langage Λ_R (et Λ_M) est uniquement un λ-calcul dans lequel certaines réduction sont interdites. En effet, supposons que si $t \in \Lambda_R$, $|t|$ désigne le terme de Λ obtenu de la manière suivante

$$\begin{array}{rcl} |x| & = & x \\ |\lambda x.t| & = & \lambda x.\, |t| \\ |t\ t'| & = & |t|\ \ |t'| \\ |\langle t\rangle| & = & |t| \\ |\mathsf{let}\ \langle x\rangle = t\ \mathsf{in}\ t'| & = & (\lambda x.\, |t'|)\ \ |t| \end{array}$$

alors on a les propriétés suivantes

Propriété 7.5.12. *Si $t, t' \in \Lambda_R$, alors $|t[x/t']| = |t|\,[x/\,|t'|]$.*

Démonstration. Par induction sur t. □

Propriété 7.5.13. *Soit $t, t' \in \Lambda_R$. Si $t \to t'$ alors $|t| \to |t'|$.*

Démonstration. Par induction sur les contextes de réduction. Les cas intéressants sont ceux pour $E = []$

— Si $(\lambda x.t)\ t' \to t[x/t']$, alors, par la Propriété 7.5.12, $|(\lambda x.t)\ t'| = (\lambda x.\,|t|)\ \ |t'| \to |t|\,[x/\,|t'|] = |t[x/t']|$.

— Si $\mathsf{let}\,\langle x\rangle = \langle t'\rangle\ \mathsf{in}\ t \to t[x/t']$, alors, par la Propriété 7.5.12, $|\mathsf{let}\,\langle x\rangle = \langle t'\rangle\ \mathsf{in}\ t| = (\lambda x.\,|t|)\ \ |t'| \to |t|\,[x/\,|t'|] = |t[x/t']|$.

□

Les règles de réduction de Λ_R forment donc un sous-ensemble des règles de réduction du λ-calcul.

Contrairement à Λ_T, Λ_R (et Λ_M) n'est pas adapté à l'évaluation partielle. En effet, par rapport au λ-calcul, il nous est juste permis de retarder l'évaluation d'un terme (Propriété 7.5.13). Donc l'exemple donné dans la Section 7.5.2.9 ne peut-être que "pire" dans Λ_R, où "pire" signifie que le nombre de réductions dans Λ_R est inférieur à celui en λ-calcul.

7.5.2.12 Précaution

Le danger de cette méthode est que l'on demande de réduire un terme qui n'est pas clos. En effet, le terme $(\mathsf{fix}\,\lambda p.\lambda n'.\mathsf{if}\ n' = 0\ \mathsf{then}\ \langle 1\rangle\ \mathsf{else}\ \langle a\times \sim (p\ (n-1))\rangle)\ n$ n'est pas clos (a est libre). Le problème de la réduction des termes non clos réside dans le fait que l'on risque donc de provoquer une erreur lors de l'exécution (en s'arrêtant sur un terme de la forme $x\ t$ par exemple où x est une variable).

Dans cette exemple, l'exécution ne rencontre pas d'erreur car la variable libre a est dans un "environnement" retardé : pour faire simple, a est entouré par $\langle\cdots\rangle$ sans échappement $\sim$ et donc sa réduction est retardée. De manière plus générale, chaque variable n'est valable que dans un seul niveau : le système de types (Section 9.6.1) devra en tenir compte.

Par contre, le terme $\sim\langle x\ t\rangle$ ne doit pas être accepté, car il se réduit en $x\ t$ qu'on ne sait pas réduire si t est une forme normale.

7.6 Conclusion

On a donné une version du calcul permettant de stopper l'exécution de terme. Chacun de ces termes gelés peut donc correspondre à l'idée générale d'une donnée, comme un terme ne pouvant se réduire.

On a vu que cette suspension du calcul pouvait aboutir à des problèmes de confluences. On a résolu ces problèmes en interdisant la substitution dans les termes gelés de termes non gelés. Cette restriction est similaire à celles de Wadler dans [Wad93] ou de Benton, Bierman, Paiva et Hyland dans [Ben+92] où les substitutions dans les termes avec promotion ne sont permises qu'avec d'autres termes avec promotion.

Le langage a les propriétés usuelles des extensions du λ-calcul (α-renommage, confluence, ...). De plus, le langage permet d'exprimer naturellement les réflexions, mais la réification est impossible.

Chapitre 8

Typage en Soft Linear Logic

On présente un système de déduction qu'on appelle logique intuitionniste avec soft-promotion (SIL), qui assigne un type à un sous-ensemble de termes de Λ_R. Il est construit à partir de la logique intuitionniste à laquelle on a ajouté la modalité ! issue de la modalité exponentielle de la logique linéaire. Dans ce chapitre, on utilise une version "soft" de la Logique Linéaire (Section 6.4).

Les *contextes de typage* sont découpés en deux : d'un coté les variables x et de l'autre les variables *gelées* (notées $\langle x \rangle$) dont on sait qu'elles sont liées à un terme gelé. On notera $p : A$ pour $x : A$ ou $\langle x \rangle : A$.

On donnera deux systèmes de types, qu'on montrera équivalents : le premier (SIL_n, Section 8.1) est en *pseudo* déduction naturelle et est inversible (chaque jugement est prouvable par au plus une dérivation). Le second (SIL_s, Section 8.3) est en calcul des séquents.

8.1 Système SIL en *pseudo* déduction naturelle

L'ensemble des *types* A est donné par

$$A \stackrel{\text{def}}{=} \alpha \mid A \to B \mid !A$$

où α désigne n'importe quel type de base.

Un *contexte de typage* $\Gamma = p_1 : A_1, \cdots, p_n : A_n$ est un ensemble d'hypothèses où toutes les variables p_i sont différentes. Le *domaine* de Γ est défini par $\mathrm{dom}(\Gamma) = \{x \mid (x : A) \in \Gamma\} \cup \{x \mid (\langle x \rangle : A) \in \Gamma\}$.

Le système de types est donné par les règles de la Figure 8.1. On dira qu'un terme $t \in \Lambda_R$ est bien typé dans SIL_n s'il existe une type A et un environnement Γ tels qu'il existe une dérivation dans ce système telle que $\Gamma \vdash_n^{\mathrm{SIL}} t : A$.

8.1.1 Exemple de typage

Le terme $\mathsf{run} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{let}\ \langle y \rangle = x\ \mathsf{in}\ y$ peut être typé dans SIL_n par la dérivation suivante.

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash_n^{\mathrm{SIL}} x : A} \qquad \textsc{Abstraction}\ \frac{\Gamma, x : A \vdash_n^{\mathrm{SIL}} t : B}{\Gamma \vdash_n^{\mathrm{SIL}} \lambda x.t : A \to B}$$

$$\textsc{Application}\ \frac{\Gamma \vdash_n^{\mathrm{SIL}} t : A \to B \qquad \Gamma \vdash_n^{\mathrm{SIL}} t' : A}{\Gamma \vdash_n^{\mathrm{SIL}} t\ t' : B}$$

$$\textsc{SoftPromotion}\ \frac{x_1 : A_1, \cdots, x_n : A_n \vdash_n^{\mathrm{SIL}} t : C}{\langle x_1 \rangle : !A_1, \cdots, \langle x_n \rangle : !A_n, y_1 : B_1, \cdots, y_m : B_m \vdash_n^{\mathrm{SIL}} \langle t \rangle : !C} \qquad \textsc{Dereliction}\ \frac{}{\Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{SIL}} x : A}$$

$$\textsc{Let}\ \frac{\Gamma \vdash_n^{\mathrm{SIL}} t' : !A \qquad \Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{SIL}} t : B}{\Gamma \vdash_n^{\mathrm{SIL}} \mathsf{let}\ \langle x \rangle = t'\ \mathsf{in}\ t : B}$$

Figure 8.1 – Système SIL en *pseudo* déduction naturelle

$$\dfrac{\dfrac{\dfrac{}{x : !A \vdash_n^{\mathrm{SIL}} x : !A} \textsc{Var} \qquad \dfrac{}{x : !A, \langle y \rangle : !A \vdash_n^{\mathrm{SIL}} y : A} \textsc{Dereliction}}{x : !A \vdash_n^{\mathrm{SIL}} \mathsf{let}\ \langle y \rangle = x\ \mathsf{in}\ y : A} \textsc{Let}}{\vdash_n^{\mathrm{SIL}} \lambda x.\, \mathsf{let}\ \langle y \rangle = x\ \mathsf{in}\ \lambda y\ : !A \to A} \textsc{Abstraction}$$

Ce système a bien la propriété d'effectivité annoncée en début de section, comme l'énonce la propriété suivante.

Propriété 8.1.1 (Inversibilité)**.** *Chaque séquent est la conclusion d'au plus une règle de dérivation*

Démonstration. Le système est presque dirigé par la syntaxe. La seule exception est lorsque t est une variable. Dans ce cas, deux règles de typage sont possibles : Dereliction et Var. Le choix d'une des deux règles est dirigé par le type de la variable x (normale ou gelée) dans Γ. Si x est normal (c'est à dire de la forme $x : A$), alors la bonne règle est la règle Var. Sinon, c'est la règle Dereliction. Comme il n'y a qu'un seul x [4] dans Γ, ce choix est unique. □

8.1.2 Règles admissibles

Dans ce système, les règles d'affaiblissement sont implicites.

Propriété 8.1.2 (Affaiblissement)**.** *Les règles*

$$\textsc{Weakening}\ \frac{\Gamma \vdash_n^{\mathrm{SIL}} t : A}{\Gamma, x : B \vdash_n^{\mathrm{SIL}} t : A} \qquad \textsc{MWeakening}\ \frac{\Gamma \vdash_n^{\mathrm{SIL}} t : A}{\Gamma, \langle x \rangle : !B \vdash_n^{\mathrm{SIL}} t : A}$$

4. par α-renommage

sont admissibles dans SIL_n.

Démonstration. Par une simple induction sur l'arbre de preuve. □

Le Lemme de Substitution est valide :

Lemme 8.1.3 (Substitution)**.** *Si* $\Gamma \vdash_n^{\mathrm{SIL}} t_1 : A$ *et* $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B$*, alors* $\Gamma \vdash_n^{\mathrm{SIL}} t_2[x/t_1] : B$*. Autrement dit, la règle*

$$\frac{\textsc{Cut}\quad \Gamma \vdash_n^{\mathrm{SIL}} t_1 : A \qquad \Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B}{\Gamma \vdash_n^{\mathrm{SIL}} t_2[x/t_1] : B}$$

est admissible.

Démonstration. Les preuves se font par induction sur t_2 et sont très semblables entre elles.

- Si $t_2 = x$, alors la dernière règle de la preuve de $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B$ est Var. De plus, $t_2[x/t_1] = t_1$ et, par hypothèse, $\Gamma \vdash_n^{\mathrm{SIL}} t_1 : A$.
- Si $t_2 = y$, alors $t_2[x/t_1] = y$. De plus, $\Gamma, x : A \vdash_n^{\mathrm{SIL}} y : B$, et par la propriété d'inversibilité, (8.1.1), $y : B$ (resp. $\langle y \rangle : !B$) est dans Γ et la dernière règle de la preuve de $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B$ est Var (resp. Dereliction), et donc $x : A$ est inutile. Donc $\Gamma \vdash_n^{\mathrm{SIL}} t_2[x/t_1]$ est dérivable par Var (resp. Dereliction).
- Les cas $t_2 = \lambda x.t$ et $t_2 = \mathsf{let}\, \langle x \rangle = t'\, \mathsf{in}\, t$ ne sont pas possibles étant donné que x n'est pas lié dans t_2.
- Si $t_2 = \lambda y.t$, alors $t_2[x/t_1] = \lambda y.t[x/t_1]$. De plus, comme $\Gamma, x : A \vdash_n^{\mathrm{SIL}} \lambda y.t : B$, par la propriété d'inversibilité (8.1.1), la dernière règle de la preuve de $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B$ est Abstraction. Donc $B = C \to D$ et $\Gamma, x : A, y : C \vdash_n^{\mathrm{SIL}} t : D$. Par hypothèse d'induction, $\Gamma, y : C \vdash_n^{\mathrm{SIL}} t[x/t_1] : D$, et donc $\Gamma \vdash_n^{\mathrm{SIL}} \lambda y.t[x/t_1] = t_2[x/t_1] : B$ en utilisant la règle Abstraction.
- Si $t_2 = \mathsf{let}\, \langle y \rangle = t'\, \mathsf{in}\, t$, alors $t_2[x/t_1] = \mathsf{let}\, \langle y \rangle = t'[x/t_1]\, \mathsf{in}\, t[x/t_1]$. De plus, comme $\Gamma, x : A \vdash_n^{\mathrm{SIL}} \mathsf{let}\, \langle y \rangle = t'\, \mathsf{in}\, t : B$, par la propriété d'inversibilité (8.1.1), la dernière règle de la preuve de $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B$ est Let. donc il existe C tel que $\Gamma, x : A, \langle y \rangle : !C \vdash_n^{\mathrm{SIL}} t : B$ et $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t' : !C$. Par hypothèse d'induction $\Gamma, \langle y \rangle : !C \vdash_n^{\mathrm{SIL}} t[x/t_1] : B$ et $\Gamma \vdash_n^{\mathrm{SIL}} t'[x/t_1] : !C$, et donc $\Gamma \vdash_n^{\mathrm{SIL}} \mathsf{let}\, \langle y \rangle = t'[x/t_1]\, \mathsf{in}\, t[x/t_1] : B$ en utilisant la règle Let.
- Si $t_2 = t\ t'$ alors $t_2[x/t_1] = t[x/t_1]\ t'[x/t_2]$. De plus $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t\ t' : B$. Par la propriété d'inversibilité (8.1.1), la dernière règle de la preuve de $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B$ est Application. Donc $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t : C \to B$ et $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t' : C$. Par hypothèse d'induction, $\Gamma \vdash_n^{\mathrm{SIL}} t[x/t_1] : C \to B$ et $\Gamma \vdash_n^{\mathrm{SIL}} t'[x/t_1] : C$ et donc $\Gamma \vdash_n^{\mathrm{SIL}} t[x/t_1]\ t'[x/t_2] = t_2[x/t_1] : B$ en utilisant le règle Application.
- Si $t_2 = \langle t \rangle$, $t_2[x/t_1] = \langle t[x/t_1] \rangle$. De plus $\Gamma, x : A \vdash_n^{\mathrm{SIL}} \langle t \rangle : B$. Donc par la propriété d'inversibilité (8.1.1), la dernière règle de la preuve de $\Gamma, x : A \vdash_n^{\mathrm{SIL}} t_2 : B$ est SoftPromotion. Donc $B = !B'$, $\Gamma = \Gamma', \langle x_i \rangle : !A_i$ et $x_i : A_i \vdash_n^{\mathrm{SIL}} t : B'$. Donc par affaiblissement (Propriété 8.1.2), $x_i : A_i, x : A \vdash_n^{\mathrm{SIL}} t : B'$. Par hypothèse d'induction, $x_i : A_i \vdash_n^{\mathrm{SIL}} t[x/t_1] : B'$, donc $\Gamma', \langle x_i \rangle : !A_i \vdash_n^{\mathrm{SIL}} \langle t \rangle [x/t] : !B'$ en utilisant la règle SoftPromotion.

□

Remarque 8.1.4. On remarque que les preuves des propriétés d'admissibilité ne changent pas la forme des preuves originales (c'est à dire qu'on ajoute pas de règles qui ne soient pas déjà là dans les preuves originales) :

— Dans l'admissibilité de l'affaiblissement, le seul changement de la preuve est d'ajouter $x : B$ ou $\langle x \rangle : !B$ tout le long de la dérivation, mais l'enchaînement des règles n'est pas modifié.

— Dans le lemme de substitution, la preuve compose les règles de $\Gamma \vdash t_1 : A$ avec la nouvelle preuve produite par hypothèse d'induction, qui a les même règles que celles de $\Gamma, x : A \vdash t_2$.

Les résultats restent donc valides *dans n'importe quel sous-système de* SIL_n. ■

Lemme 8.1.5 (Méta Substitution)**.** *Si* $\Gamma \vdash_n^{\mathrm{SIL}} \langle t_1 \rangle : !A$ *et* $\Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{SIL}} t_2 : B$ *alors* $\Gamma \vdash_n^{\mathrm{SIL}} t_2[x/t_1] : B$*. Autrement dit, la règle*

$$\frac{\begin{array}{c}\textsc{MCut}\\ \Gamma \vdash_n^{\mathrm{SIL}} \langle t_1 \rangle : !A \qquad \Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{SIL}} t_2 : B\end{array}}{\Gamma \vdash_n^{\mathrm{SIL}} t_2[x/t_1] : B}$$

est admissible.

Démonstration. La preuve est semblable à celle du lemme de substitution, par induction sur t_2. Les seuls cas différents sont pour $t_2 = x$ et $t_2 = \langle t \rangle$. Dans ces cas on notera que $\Gamma \vdash \langle t_1 \rangle : !A$ implique, par inversibilité qu'il existe $\Delta = x_1 : A_1, \cdots, x_n : A_n$ tel que $\Delta \vdash t_1 : A$ et $\Gamma = \Gamma', \langle \Delta \rangle$ et Γ' ne contient pas $\langle y \rangle$.

— Si $t_2 = x$, alors, par inversibilité, la dernière règle de la preuve de $\Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{SIL}} x : A$ est Dereliction. De plus, on doit prouver que $\Gamma \vdash_n^{\mathrm{SIL}} t_1 : A$. On sait que $\Delta \vdash_n^{\mathrm{SIL}} t_1 : A$ dans le système SIL_n. Posons $\Delta' = y_1 : A_1, \cdots, y_n : A_n$. Donc par affaiblissement, $\Delta, \Delta' \vdash_n^{\mathrm{SIL}} t_1 : A$. De plus, par déréliction, $\langle \Delta' \rangle \vdash_n^{\mathrm{SIL}} y_i : A_i$. Pour, par admissibilité de la coupure sur les variables x_i de Δ, on a $\langle \Delta' \rangle \vdash_n^{\mathrm{SIL}} t_i[x_i/y_i] : A$. Donc, comme $\Delta' = \Delta[x_i/y_i]$, alors $\langle \Delta \rangle \vdash_n^{\mathrm{SIL}} t_1 : A$ Il s'en suit que $\Gamma', \langle \Delta \rangle \vdash_n^{\mathrm{SIL}} t : A$.

— Si $t_2 = \langle t \rangle$, alors par propriété d'inversibilité la dernière règle de la preuve de $\Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{SIL}} \langle t \rangle : !B$ est SoftPromotion. On doit prouver que $\Gamma \vdash_n^{\mathrm{SIL}} \langle t[x/t_1] \rangle : !B$. On sait que $x_i : A_i \vdash_n^{\mathrm{SIL}} t_1 : A$ par inversion de SoftPromotion. De plus, on sait que $\Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{SIL}} \langle t \rangle : !B$ ce qui implique que $x_i : A_i, x : A \vdash_n^{\mathrm{SIL}} t : B$. Donc, par le lemme de substitution, $x_i : A_i \vdash_n^{\mathrm{SIL}} t[x/t_1] : B$. Donc, par SoftPromotion et Weakening, $\Gamma \vdash_n^{\mathrm{SIL}} \langle t[x/t_1] \rangle : !B$.

□

Remarque 8.1.6. Dans le même esprit que la Remarque 8.1.4, la preuve d'admissibilité n'introduit pas de nouvelle règle dans la dérivation. Donc ce résultat reste lui aussi valide *dans n'importe quel sous-système de* SIL_n. ■

8.2 Correction du système de types

Le système de types est cohérent avec la définition de variables libres.

Propriété 8.2.1. *Si* $\Gamma \vdash_n^{\mathrm{SIL}} t : A$, *alors*

— *si* $x \in \mathrm{FV}^{\mathsf{let}}(t)$ *alors* $\langle x \rangle : !B \in \Gamma$

— $\mathrm{FV}(t) \subset \mathrm{dom}(\Gamma)$

Démonstration. Par induction sur t. □

On notera que si $x \in \mathrm{FV}^{\lambda}(t)$, alors $x : B$ n'est pas nécessairement dans Γ à cause de la règle Dereliction : $\langle x \rangle : !B \vdash_n^{\mathrm{SIL}} x : B$.

Lemme 8.2.2 (Réduction en tête)**.** *Si* $t \to_{\mathrm{head}} t'$ *et* $\Gamma \vdash_n^{\mathrm{SIL}} t : A$, *alors* $\Gamma \vdash_n^{\mathrm{SIL}} t' : A$

Démonstration. Par analyse des cas de t.

— Si $t = (\lambda x.t_2)t_1$, alors $t' = t_2[x/t_1]$. Si $\Gamma \vdash_n^{\mathrm{SIL}} (\lambda x.t_2)t_1 : A$, par inversibilité, $\Gamma, x : B \vdash_n^{\mathrm{SIL}} t_2 : A$ et $\Gamma \vdash_n^{\mathrm{SIL}} t_1 : B$. Par le Lemme de Substitution 8.1.3, $\Gamma \vdash_n^{\mathrm{SIL}} t' : A$.

— Si $t = \mathsf{let}\, \langle x \rangle = \langle t_1 \rangle\, \mathsf{in}\, t_2$, alors $t' = t_2[x/t_1]$. Si $\Gamma \vdash_n^{\mathrm{SIL}} \mathsf{let}\, \langle x \rangle = \langle t_1 \rangle\, \mathsf{in}\, t_2 : A$, par inversibilité, alors $\Gamma, \langle x \rangle : !B \vdash_n^{\mathrm{SIL}} t_2 : A$ et $\Gamma \vdash_n^{\mathrm{SIL}} \langle t_1 \rangle : !B$. Par le deuxième Lemme de Substitution 8.1.5, $\Gamma \vdash_n^{\mathrm{SIL}} t' : A$

□

Lemme 8.2.3 (Compositionalité)**.** *Si* $\Gamma \vdash_n^{\mathrm{SIL}} E[t] : A$, *il existe* $\Gamma' \supseteq \Gamma$ *et* B *tels que*

— $\Gamma' \vdash_n^{\mathrm{SIL}} t : B$,

— *pour tout* t', *si* $\Gamma' \vdash_n^{\mathrm{SIL}} t' : B$ *alors* $\Gamma \vdash_n^{\mathrm{SIL}} E[t'] : A$

Démonstration. Par induction sur E.

— Si $E = []$, alors $\Gamma' \stackrel{\mathrm{def}}{=} \Gamma$ et $B \stackrel{\mathrm{def}}{=} A$ fonctionne.

— Si $E = E'\ t_1$ (resp. $E = t_1\ E'$), alors par inversibilité, $\Gamma \vdash_n^{\mathrm{SIL}} E'[t] : C \to A$ et $\Gamma \vdash_n^{\mathrm{SIL}} t_1 : C$ (resp. $\Gamma \vdash_n^{\mathrm{SIL}} E'[t] : C$ et $\Gamma \vdash_n^{\mathrm{SIL}} t_1 : C \to A$). Par hypothèse d'induction, il existe $\Gamma' \supseteq \Gamma$ et B tels que $\Gamma' \vdash_n^{\mathrm{SIL}} t : B$. De plus, pour tout t' tel que $\Gamma' \vdash_n^{\mathrm{SIL}} t' : B$, $\Gamma \vdash_n^{\mathrm{SIL}} E'[t'] : C \to A$ (resp. $\Gamma \vdash_n^{\mathrm{SIL}} E'[t'] : C$). Donc $\Gamma \vdash_n^{\mathrm{SIL}} E[t'] : A$ par la règle Application.

— Si $E = \lambda x.E'$, alors, par inversibilité, $A = C \to D$ et $\Gamma, x : C \vdash_n^{\mathrm{SIL}} E'[t] : D$. Par hypothèse d'induction, il existe B et $\Gamma' \supseteq \Gamma$ tels que $\Gamma', x : C \vdash_n^{\mathrm{SIL}} t : B$. De plus, pour tout t' tel que $\Gamma', x : C \vdash_n^{\mathrm{SIL}} t' : B$, $\Gamma, x : C \vdash_n^{\mathrm{SIL}} E'[t'] : D$. Donc $\Gamma \vdash_n^{\mathrm{SIL}} E[t'] : A$ par la règle Abstraction.

— Si $E = \mathsf{let}\, \langle x \rangle = t_1\, \mathsf{in}\, E'$ (resp. $E = \mathsf{let}\, \langle x \rangle = E'\, \mathsf{in}\, t_1$), alors, par inversibilité, il existe C tel que $\Gamma, \langle x \rangle : !C \vdash_n^{\mathrm{SIL}} E'[t] : A$ et $\Gamma \vdash_n^{\mathrm{SIL}} t_1 : !C$ (resp. $\Gamma, \langle x \rangle : !C \vdash_n^{\mathrm{SIL}} t_1 : A$ et $\Gamma \vdash_n^{\mathrm{SIL}} E'[t] : !C$). Par hypothèse d'induction, il existe B et $\Gamma' \supseteq \Gamma$ tels que $\Gamma', \langle x \rangle : !C \vdash_n^{\mathrm{SIL}} t : B$ (resp. $\Gamma' \vdash_n^{\mathrm{SIL}} t : B$). De plus, pour tout t' tel que $\Gamma', x : C \vdash_n^{\mathrm{SIL}} t' : B$ (resp. $\Gamma' \vdash_n^{\mathrm{SIL}} t' : B$), $\Gamma, x : C \vdash_n^{\mathrm{SIL}} E'[t'] : A$ (resp. $\Gamma \vdash_n^{\mathrm{SIL}} E'[t'] : !C$). Alors $\Gamma \vdash_n^{\mathrm{SIL}} E[t'] : A$ par la règle Let.

□

Remarque 8.2.4. De la même manière que les Remarques 8.1.4 et 8.1.6, les arbres de preuves construits ici utilisent les mêmes règles que les arbres originaux. Donc, à nouveau, le Lemme de Composition est toujours dans *n'importe quel sous-système de* SIL_n. ■

Finalement, on obtient le résultat de correction du système.

Théorème 8.2.5 (Réduction du sujet)**.** *Si* $\Gamma \vdash_n^{\mathrm{SIL}} t : A$ *et* $t \to t'$ *alors* $\Gamma \vdash_n^{\mathrm{SIL}} t' : A$.

Démonstration. Si $t \to t'$, il existe E, s, s' tel que $t = E[s]$, $s \to_{\mathrm{head}} s'$ et $t' = E[s']$. Par le Lemme de Composition, $\Gamma' \vdash_n^{\mathrm{SIL}} s : B$. Par le Lemme 8.2.2, $\Gamma' \vdash_n^{\mathrm{SIL}} s' : B$. Par compositionnalité à nouveau, $\Gamma \vdash_n^{\mathrm{SIL}} E[s'] = t' : A$. □

Remarque 8.2.6. Notre système évite un problème courant du Théorème de Réduction du Sujet pour les syntaxes de la Logique Linéaire (par exemple dans [RR97 ; GRR07]). Ce problème vient de la gestion linaire des ressourses : on ne peut pas dupliquer de terme qui ne sont pas de la forme $\langle t \rangle$, car sinon, on risque de perdre le bon typage. Notre système échappe à ce problème cas nous n'avons pas une gestion linéaire des variables (les règles d'affaiblissement et de contraction peuvent être appliquées à des variables de n'importe quel type). ■

Remarque 8.2.7. Selon la Remarque 8.2.4 for pour le Lemme de Compositionnalité et les Remarques 8.1.4 et 8.1.6 pour les Lemmes de Substitution, le résultat du Théorème de Réduction du Sujet est toujours valide dans *tout sous-système de* SIL_n. ■

Théorème 8.2.8 (Normalisation forte)**.** *Si* $\Gamma \vdash_n^{\mathrm{SIL}} t : A$ *alors tout chemin de réduction de* t *est fini.*

Démonstration. En oubliant les notations nouvelles (modalité !, termes gelés $\langle t \rangle$ et lieurs let) on obtient le λ-calcul simplement typé, et toute réduction dans notre langage correspond à au moins une réduction en λ-calcul (Propriété 7.5.13). Comme le λ-calcul simplement typé est fortement normalisable, il en va de même pour notre langage. □

8.3 Typage en calcul des séquents

Dans cette section on donne une présentation sous forme de calcul des séquents de notre système de types. Ce système est appelé SIL_s. Il s'inspire de la logique linéaire pour la gestion de la modalité, mais ne traite pas de la gestion des ressources. Les règles de SIL_s sont présentées dans la Figure 8.2. La règle d'élimination de la modalité ! est la règle Dereliction. Elle correspond au processus reflexif transformant un terme gelé (c'est à dire une donnée) en terme réductible (c'est à dire un programme). Les variables gelées ne sont jamais liées par le lieur λ mais par le lieur let. Ce lieur permet d'un coté d'utiliser une variable gelée et d'un autre coté d'introduire une nouvelle variable y qui elle pourra être liée par un λ. Le programme run donné plus tôt en est une illustration.

La règle Let est la même que celle utilisée par Wadler dans [Wad93] lorsqu'il cherche à donner une syntaxe cohérente à la logique linéaire, c'est à dire que toutes les preuves du typage d'un même terme aient la même sémantique. En effet, la règle Cut combinée

$$\textsc{Let}\ \frac{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{SIL}} t : B}{\Gamma, y : !A \vdash_s^{\mathrm{SIL}} \mathsf{let}\ \langle x\rangle = y\ \mathsf{in}\ t : B} \qquad \textsc{Axiom}\ \frac{}{x : A \vdash_s^{\mathrm{SIL}} x : A}$$

$$\textsc{Weakening}\ \frac{\Gamma \vdash_s^{\mathrm{SIL}} t : B}{\Gamma, x : A \vdash_s^{\mathrm{SIL}} t : B} \qquad \textsc{MWeakening}\ \frac{\Gamma \vdash_s^{\mathrm{SIL}} t : B}{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{SIL}} t : B}$$

$$\textsc{Contraction}\ \frac{\Gamma, y : A, z : A \vdash_s^{\mathrm{SIL}} t : C}{\Gamma, x : A \vdash_s^{\mathrm{SIL}} t[y/x][z/x] : C} \qquad \textsc{MContraction}\ \frac{\Gamma, \langle y\rangle : !A, \langle z\rangle : !A \vdash_s^{\mathrm{SIL}} t : C}{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{SIL}} t[y/x][z/x] : C}$$

$$\rightarrow\textsc{Left}\ \frac{\Gamma, x : B \vdash_s^{\mathrm{SIL}} t : C \qquad \Delta \vdash_s^{\mathrm{SIL}} t' : A}{\Gamma, \Delta, f : A \to B \vdash_s^{\mathrm{SIL}} t[x/f\ t'] : C} \qquad \rightarrow\textsc{Right}\ \frac{\Gamma, x : A \vdash_s^{\mathrm{SIL}} t : B}{\Gamma \vdash_s^{\mathrm{SIL}} \lambda x.t : A \to B}$$

$$\textsc{SoftPromotion}\ \frac{x_1 : A_1, \cdots, x_n : A_n \vdash_s^{\mathrm{SIL}} t : B}{\langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n \vdash_s^{\mathrm{SIL}} \langle t\rangle : !B} \qquad \textsc{Dereliction}\ \frac{\Gamma, x : A \vdash_s^{\mathrm{SIL}} t : B}{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{SIL}} t : B}$$

$$\textsc{Cut}\ \frac{\Gamma \vdash_s^{\mathrm{SIL}} t' : A \quad \Delta, x : A \vdash_s^{\mathrm{SIL}} t : B}{\Gamma, \Delta \vdash_s^{\mathrm{SIL}} t[x/t'] : B}$$

FIGURE 8.2 – règles de SIL_s

à la règle PROMOTION introduit une incohérence si l'élimination des coupures est faite sans précaution. La solution adoptée par WADLER est la même que la nôtre, c'est à dire éliminer la coupure uniquement lorsque les termes à substituer sont eux même issus d'une règle PROMOTION.

La situation est similaire dans le système SIL_s. La règle LET contrôle les substitutions des termes t de type $!A$. Elles ne peuvent s'effectuer que lorsque t est de la forme $\langle t' \rangle$.

8.4 Équivalence des deux systèmes

Le théorème suivant énonce l'équivalence de SIL_n et SIL_s.

Théorème 8.4.1. *Le jugement* $\Gamma \vdash_n^{\mathrm{SIL}} t : A$ *est dérivable si et seulement si* $\Gamma \vdash_s^{\mathrm{SIL}} t : A$ *est dérivable.*

Démonstration. Ce résultat est montré en donnant les traductions $\mathcal{N}$ et $\mathcal{S}$, passant de la version en calcul des séquents à celle en déduction naturelle, et réciproquement. Ces traductions sont données dans les deux sections suivantes. □

8.4.1 Du calcul des séquents à la déduction naturelle

On définit une traduction vers les preuves en déduction naturelle, notée $\mathcal{N}$. Une preuve en calcul des séquents est notée π.

— La preuve de l'axiome n'est pas changée

— la preuve

$$\rightarrow\textsc{Right}\ \frac{\begin{array}{c}\pi\\ \Gamma, x : A \vdash_s^{\mathrm{SIL}} t : B\end{array}}{\Gamma \vdash_s^{\mathrm{SIL}} \lambda x.t : A \rightarrow B}$$

est changée en

$$\textsc{Abstraction}\ \frac{\begin{array}{c}\mathcal{N}(\pi)\\ \Gamma, x : A \vdash_n^{\mathrm{SIL}} t : B\end{array}}{\Gamma \vdash_n^{\mathrm{SIL}} \lambda x.t : A \rightarrow B}$$

— La règle

$$\textsc{Let}\ \frac{\begin{array}{c}\pi\\ \Gamma, \langle x \rangle :!A \vdash_s^{\mathrm{SIL}} t : B\end{array}}{\Gamma, y :!A \vdash_s^{\mathrm{SIL}} \mathsf{let}\ \langle x \rangle = y\ \mathsf{in}\ t : B}$$

est changée en

$$\textsc{Let}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma, y :!A \vdash_n^{\mathrm{SIL}} y :!A} \qquad \textsc{Weakening}\ \dfrac{\overset{\mathcal{N}(\pi)}{\Gamma, \langle x\rangle :!A \vdash_n^{\mathrm{SIL}} t : B}}{\Gamma, y :!A, \langle x\rangle :!A \vdash_n^{\mathrm{SIL}} t : B}}{\Gamma, y :!A \vdash_n^{\mathrm{SIL}} \mathsf{let}\ \langle x\rangle = y\ \mathsf{in}\ t : B}$$

en utilisant l'admissibilité de la règle Weakening en déduction naturelle (Propriété 8.1.2).

— La règle

$$\textsc{Cut}\ \dfrac{\overset{\pi_1}{\Gamma \vdash_s^{\mathrm{SIL}} t' : A} \qquad \overset{\pi_2}{\Delta, x : A \vdash_s^{\mathrm{SIL}} t : B}}{\Gamma, \Delta \vdash_s^{\mathrm{SIL}} t[x/t'] : B}$$

est changée en

$$\textsc{Cut}\ \dfrac{\textsc{Weakening}\ \dfrac{\overset{\mathcal{N}(\pi_1)}{\Gamma \vdash_n^{\mathrm{SIL}} t' : A}}{\Gamma, \Delta \vdash_n^{\mathrm{SIL}} t' : A} \qquad \textsc{Weakening}\ \dfrac{\overset{\mathcal{N}(\pi_2)}{\Delta, x : A \vdash_n^{\mathrm{SIL}} t : B}}{\Gamma, \Delta, x : A \vdash_n^{\mathrm{SIL}} t : B}}{\Gamma, \Delta \vdash_n^{\mathrm{SIL}} t[x/t'] : B}$$

en utilisant l'admissibilité de la règle Cut en déduction naturelle (Lemme 8.1.3)

— La règle

$$\rightarrow\textsc{Left}\ \dfrac{\overset{\pi_1}{\Gamma \vdash_s^{\mathrm{SIL}} t' : A} \qquad \overset{\pi_2}{\Delta, x : B \vdash_s^{\mathrm{SIL}} t : C}}{\Gamma, \Delta, y : A \to B \vdash_s^{\mathrm{SIL}} t[x/y\ t'] : B}$$

est changée en

$$\textsc{Application}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma, y : A \to B \vdash_n^{\mathrm{SIL}} y : A \to B} \qquad \textsc{Weakening}\ \dfrac{\overset{\mathcal{N}(\pi_1)}{\Gamma \vdash_n^{\mathrm{SIL}} t' : A}}{\Gamma, y : A \to B \vdash_n^{\mathrm{SIL}} t' : A}}{\textsc{Cut}\ \dfrac{\textsc{Weakening}\ \dfrac{\Gamma, y : A \to B \vdash_n^{\mathrm{SIL}} y\ t' : B}{\Gamma, \Delta, y : A \to B \vdash_n^{\mathrm{SIL}} y\ t' : B} \qquad \textsc{Weakening}\ \dfrac{\overset{\mathcal{N}(\pi_2)}{\Delta, x : B \vdash_n^{\mathrm{SIL}} t : C}}{\Gamma, \Delta, x : B \vdash_n^{\mathrm{SIL}} t : C}}{\Gamma, \Delta, y : A \to B \vdash_n^{\mathrm{SIL}} t[x/y\ t'] : B}}$$

en utilisant les admissibilités des règles Cut (Lemme 8.1.3) Weakening (Propriété 8.1.2) en déduction naturelle.

— La règle

$$\textsc{SoftPromotion}\ \dfrac{\overset{\pi}{x_i : A_i \vdash_s^{\mathrm{SIL}} t : B}}{\langle x_i\rangle :!A_i \vdash_s^{\mathrm{SIL}} \langle t\rangle :!B}$$

est changée en

$$\textsc{SoftPromotion}\ \frac{\begin{array}{c}\mathcal{N}(\pi)\\ x_i : A_i \vdash_n^{\mathrm{SIL}} t : B\end{array}}{\langle x_i\rangle :!A_i \vdash_n^{\mathrm{SIL}} \langle t\rangle :!B}$$

— La règle

$$\textsc{Dereliction}\ \frac{\begin{array}{c}\pi\\ \Gamma, x : A \vdash_s^{\mathrm{SIL}} t : B\end{array}}{\Gamma, \langle x\rangle :!A \vdash_s^{\mathrm{SIL}} t : B}$$

est changée en

$$\textsc{Cut}\ \frac{\textsc{Dereliction}\ \dfrac{}{\Gamma, \langle y\rangle :!A \vdash_n^{\mathrm{SIL}} y : A} \qquad \textsc{MWeakening}\ \dfrac{\begin{array}{c}\mathcal{N}(\pi)\\ \Gamma, x : A \vdash_n^{\mathrm{SIL}} t : B\end{array}}{\Gamma, \langle y\rangle :!A, x : A \vdash_n^{\mathrm{SIL}} t : B}}{\Gamma, \langle y\rangle :!A \vdash_n^{\mathrm{SIL}} t[x/y] = t : B}$$

en utilisant les admissibilités des règles Cut (Lemme 8.1.3) MWeakening (Propriété 8.1.2) en déduction naturelle.

— La règle

$$\textsc{Contraction}\ \frac{\begin{array}{c}\pi\\ \Gamma, y : A, z : A \vdash_s^{\mathrm{SIL}} t : B\end{array}}{\Gamma, x : A \vdash_s^{\mathrm{SIL}} t[y/x][z/x] : B}$$

est changée en

$$\textsc{Cut}\ \frac{\textsc{Var}\ \dfrac{}{\Gamma, x : A \vdash_n^{\mathrm{SIL}} x : A} \qquad \textsc{Cut}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma, x : A, z : A \vdash_n^{\mathrm{SIL}} x : A} \qquad \textsc{Weakening}\ \dfrac{\begin{array}{c}\mathcal{N}(\pi)\\ \Gamma, z : A, y : A \vdash_n^{\mathrm{SIL}} t : B\end{array}}{\Gamma, x : A, z : A, y : A \vdash_n^{\mathrm{SIL}} t : B}}{\Gamma, x : A, z : A \vdash_n^{\mathrm{SIL}} t[y/x] : B}}{\Gamma, x : A \vdash_n^{\mathrm{SIL}} t[y/x][z/x] : B}$$

— La règle

$$\textsc{MContraction}\ \frac{\begin{array}{c}\pi\\ \Gamma, \langle y\rangle :!A, \langle z\rangle :!A \vdash_s^{\mathrm{SIL}} t : B\end{array}}{\Gamma, \langle x\rangle :!A \vdash_s^{\mathrm{SIL}} t[y/x][z/x] : B}$$

est changée en

$$\textsc{MCut}\ \frac{\textsc{MWeakening}\ \dfrac{\textsc{Cut}\ \dfrac{\begin{array}{c}\mathcal{N}(\pi)\\ \Gamma, \langle z\rangle :!A, \langle y\rangle :!A \vdash_n^{\mathrm{SIL}} t : B\end{array}}{\Gamma, \langle x\rangle :!A, \langle z\rangle :!A, \langle y\rangle :!A \vdash_n^{\mathrm{SIL}} t : B} \qquad \dfrac{\vdots}{\Gamma, \langle x\rangle :!A, \langle z\rangle :!A \vdash_n^{\mathrm{SIL}} \langle x\rangle :!A}}{\Gamma, \langle x\rangle :!A, \langle z\rangle :!A \vdash_n^{\mathrm{SIL}} t[y/x] : B} \qquad \dfrac{\vdots}{\Gamma, \langle x\rangle :!A \vdash_n^{\mathrm{SIL}} \langle x\rangle :!A}}{\Gamma, \langle x\rangle :!A \vdash_n^{\mathrm{SIL}} t[y/x][z/x] : B}$$

8.4.2 De la déduction naturelle au calcul des séquents

On note $\mathcal{S}$ une traduction vers les calculs des séquents. Dans cette section une preuve en déduction naturelle est notée π.

— La preuve

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash_n^{\text{SIL}} x : A}$$

est changée en

$$\textsc{Weakening}\ \frac{\textsc{Axiom}\ \dfrac{}{x : A \vdash_s^{\text{SIL}} x : A}}{\Gamma, x : A \vdash_s^{\text{SIL}} x : A}$$

— La preuve

$$\textsc{Abstraction}\ \frac{\overset{\pi}{\Gamma, x : A \vdash_n^{\text{SIL}} t : B}}{\Gamma \vdash_n^{\text{SIL}} \lambda x.t : A \to B}$$

est changée en

$$\to\textsc{Right}\ \frac{\overset{\mathcal{S}(\pi)}{\Gamma, x : A \vdash_s^{\text{SIL}} t : B}}{\Gamma \vdash_s^{\text{SIL}} \lambda x.t : A \to B}$$

— La preuve

$$\textsc{Let}\ \frac{\overset{\pi_1}{\Gamma, \vdash_n^{\text{SIL}} t' :!A} \qquad \overset{\pi_2}{\Gamma, \langle x \rangle :!A \vdash_n^{\text{SIL}} t : B}}{\Gamma \vdash_n^{\text{SIL}} \mathsf{let}\ \langle x \rangle = t'\ \mathsf{in}\ t : B}$$

est changée en

$$\textsc{Contraction}\ \frac{\textsc{Cut}\ \dfrac{\overset{\mathcal{S}(\pi_1)}{\Gamma \vdash_s^{\text{SIL}} t' :!A} \qquad \dfrac{\overset{\mathcal{S}(\pi_2)}{\Gamma, \langle x \rangle :!A \vdash_s^{\text{SIL}} t : B}}{\Gamma, y :!A \vdash_s^{\text{SIL}} \mathsf{let}\ \langle x \rangle = y\ \mathsf{in}\ t : B}\ \textsc{Let}}{\Gamma, \Gamma \vdash_s^{\text{SIL}} \mathsf{let}\ \langle x \rangle = t'\ \mathsf{in}\ t : B}}{\Gamma \vdash_s^{\text{SIL}} \mathsf{let}\ \langle x \rangle = t'\ \mathsf{in}\ t : B}$$

— La preuve

$$\textsc{Application}\ \frac{\overset{\pi_1}{\Gamma \vdash_n^{\text{SIL}} t : A \to B} \qquad \overset{\pi_2}{\Gamma \vdash_n^{\text{SIL}} t' : A}}{\Gamma \vdash_n^{\text{SIL}} t\ t' : B}$$

est changée en

$$\textsc{Contraction}\ \frac{\textsc{Cut}\ \dfrac{\mathcal{S}(\pi_1)\quad \Gamma \vdash_s^{\mathrm{SIL}} t : A \to B \qquad \dfrac{\mathcal{S}(\pi_2)\quad \Gamma \vdash_s^{\mathrm{SIL}} t' : A \qquad \dfrac{}{x : B \vdash_s^{\mathrm{SIL}} x : B}\ \textsc{Var}}{\Gamma, y : A \to B \vdash_s^{\mathrm{SIL}} x[x/y\ t'] : B}\ {\to}\textsc{Left}}{\Gamma, \Gamma \vdash_s^{\mathrm{SIL}} (y\ t')[y/t] : B}}{\Gamma \vdash_s^{\mathrm{SIL}} (y\ t')[y/t] : B}$$

où y n'est pas liée dans Γ ou t' et $x \neq y$ n'apparaît pas dans Γ.

— La preuve

$$\textsc{SoftPromotion}\ \frac{\overset{\pi}{x_i : A_i \vdash_n^{\mathrm{SIL}} t : B}}{\Gamma, \langle x_i \rangle :! A_i \vdash_n^{\mathrm{SIL}} \langle t \rangle :! B}$$

est changée en

$$\textsc{SoftPromotion}\ \frac{\textsc{Weakening}\ \dfrac{\overset{\mathcal{S}(\pi)}{x_i : A_i \vdash_s^{\mathrm{SIL}} t : B}}{\langle x_i \rangle :! A_i \vdash_s^{\mathrm{SIL}} \langle t \rangle :! B}}{\Gamma, \langle x_i \rangle :! A_i \vdash_s^{\mathrm{SIL}} \langle t \rangle :! B}$$

— La preuve

$$\textsc{Dereliction}\ \frac{}{\Gamma, \langle x \rangle :! A \vdash_n^{\mathrm{SIL}} x : A}$$

est changée en

$$\textsc{Dereliction}\ \frac{\textsc{Weakening}\ \dfrac{\textsc{Axiom}\ \dfrac{}{x : A \vdash_s^{\mathrm{SIL}} x : A}}{\Gamma, x : A \vdash_s^{\mathrm{SIL}} x : A}}{\Gamma, \langle x \rangle :! A \vdash_s^{\mathrm{SIL}} x : A}$$

Dans ce qui suit, on dira que $\Gamma \vdash^{\mathrm{SIL}} t : A$ dans SIL lorsque $\Gamma \vdash_n^{\mathrm{SIL}} t : A$ SIL_n (ou, de manière équivalente, lorsque $\Gamma \vdash_s^{\mathrm{SIL}} t : A$ in SIL_s).

8.5 Conclusion

On a donné un premier système de types à Λ_R. On utilise deux systèmes équivalents : l'un en pseudo déduction naturelle, et l'autre en calcul des séquents. Ce système est inspiré de la Soft Linear Logic. Il est correct en regarde de la sémantique opérationnelle de Λ_R (on terme bien typé le reste malgré ses réductions).

Encore une fois, le problème de la cohérence de la syntaxe de la logique linéaire reste présent dans notre langage, car notre système de types subit les même contraintes que celui de Wadler, Benton, ...

Dans le chapitre suivant on va exprimer notre langage dans une système de type strictement plus large en utilisant toute la puissance de l'exponentiel de la Logique Linéaire.

Chapitre 9

Typage avec promotion

Dans ce chapitre on propose un autre système de types aux termes de Λ_R, inspiré lui de la logique linéaire (avec Promotion). Le principal changement qu'il introduit est la possibilité de typer le digging $\lambda X.\, \mathsf{let}\, \langle x \rangle = X\, \mathsf{in}\, \langle\langle x \rangle\rangle$. Il est aussi similaire aux systèmes de types inspirés de la logique modale [DP96], comme on le verra plus tard.

De la même manière que dans le Chapitre 8, on va donner deux systèmes, un en *pseudo* déduction naturelle (PIL_n, Section 9.1) et l'autre en calcul des séquents (PIL_s, Section 9.3).

On montrera entre autres que l'ensemble des termes de Λ_R typés par PIL est strictement plus grand que celui des termes typés par SIL (Chapitre 10).

9.1 Système PIL en *pseudo* déduction naturelle

L'ensemble des types est le même que pour SIL :

$$A \stackrel{\text{def}}{=} \alpha \mid A \to B \;\mid\; !A$$

où α désigne n'importe quel type de base. Le système en *pseudo* déduction naturelle est donné par les règles suivantes de la Figure 9.1. Ce système utilise également des environnements de typage contenant des hypothèses de la forme $x : A$ ou $\langle x \rangle : !A$. Les jugements sont de la forme $\Gamma \vdash_n^{\mathrm{PIL}} t : A$.

9.1.1 Exemple de typage

Le terme $\mathsf{dig} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{let}\, \langle y \rangle = x\, \mathsf{in}\, \langle\langle y \rangle\rangle$ peut être typé dans PIL_n.

$$\dfrac{\dfrac{\text{Var}\ \dfrac{}{x : !A \vdash_n^{\mathrm{PIL}} x : !A} \qquad \dfrac{\dfrac{\dfrac{}{\langle y \rangle : !A \vdash_n^{\mathrm{PIL}} y : A}\ \text{Dereliction}}{\langle y \rangle : !A \vdash_n^{\mathrm{PIL}} \langle y \rangle : !A}\ \text{Promotion}}{x : !A, \langle y \rangle : !A \vdash_n^{\mathrm{PIL}} \langle\langle y \rangle\rangle : !!A}\ \text{Promotion}}{x : !A \vdash_n^{\mathrm{PIL}} \mathsf{let}\, \langle y \rangle = x\, \mathsf{in}\, \langle\langle y \rangle\rangle : !!A}\ \text{Let}}{\vdash_n^{\mathrm{PIL}} \lambda x.\, \mathsf{let}\, \langle y \rangle = x\, \mathsf{in}\, \langle\langle y \rangle\rangle : !A \to !!A}\ \text{Abstraction}$$

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash_n^{\mathrm{PIL}} x : A} \qquad \textsc{Abstraction}\ \frac{\Gamma, x : A \vdash_n^{\mathrm{PIL}} t : B}{\Gamma \vdash_n^{\mathrm{PIL}} \lambda x.t : A \to B}$$

$$\textsc{Application}\ \frac{\Gamma \vdash_n^{\mathrm{PIL}} t : A \to B \qquad \Gamma \vdash_n^{\mathrm{PIL}} t' : A}{\Gamma \vdash_n^{\mathrm{PIL}} t\ t' : B}$$

$$\textsc{Promotion}\ \frac{\langle x_1 \rangle : !A_1, \cdots, \langle x_n \rangle : !A_n \vdash_n^{\mathrm{PIL}} t : C}{\langle x_1 \rangle : !A_1, \cdots, \langle x_n \rangle : !A_n, y_1 : B_1, \cdots, y_m : B_m \vdash_n^{\mathrm{PIL}} \langle t \rangle : !C} \qquad \textsc{Dereliction}\ \frac{}{\Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{PIL}} x : A}$$

$$\textsc{Let}\ \frac{\Gamma \vdash_n^{\mathrm{PIL}} t' : !A \qquad \Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{PIL}} t : B}{\Gamma \vdash_n^{\mathrm{PIL}} \mathsf{let}\ \langle x \rangle = t'\ \mathsf{in}\ t : B}$$

Figure 9.1 – Système PIL en *pseudo* déduction naturelle

Ce système a la même propriété d'effectivité que SIL_n, comme l'énonce la propriété suivante.

Propriété 9.1.1 (Inversibilité)**.** *Chaque séquent est la conclusion d'au plus une règle de dérivation.*

Démonstration. Même démonstration que pour la Propriété 8.1.1 (exactement). Le système est presque dirigé par la syntaxe. La seule exception est lorsque t est une variable. Dans ce cas, deux règles de typage sont possibles : Dereliction et Var. Le choix d'une des deux règles est dirigé par le type de la variable x (normale ou gelée) dans Γ. Si x est normal (c'est à dire de la forme $x : A$), alors la bonne règle est la règle Var. Sinon, c'est la règle Dereliction. Comme il n'y a qu'un seul x [5] dans Γ, ce choix est unique. □

9.1.2 Règles admissibles

Les propriétés d'affaiblissement sont toujours vérifiée.

Propriété 9.1.2 (Affaiblissement)**.** *Les règles*

$$\textsc{Weakening}\ \frac{\Gamma \vdash_n^{\mathrm{PIL}} t : A}{\Gamma, x : B \vdash_n^{\mathrm{PIL}} t : A} \qquad \textsc{MWeakening}\ \frac{\Gamma \vdash_n^{\mathrm{PIL}} t : A}{\Gamma, \langle x \rangle : !B \vdash_n^{\mathrm{PIL}} t : A}$$

sont admissibles dans PIL_n.

Démonstration. Même démonstration que pour la Propriété 8.1.2 □

5. par α-renommage

Propriété 9.1.3. *Si* $\Gamma \vdash_n^{\text{PIL}} \langle t\rangle : !A$, *alors* $\Gamma \vdash_n^{\text{PIL}} \langle\langle t\rangle\rangle : !!A$ *et* $\Gamma \vdash_n^{\text{PIL}} t : A$.

Démonstration. Si $\Gamma = \langle\Gamma_1\rangle, \Gamma_2$ et que Γ_2 ne comporte pas de variables $\langle x\rangle$, alors $\langle\Gamma_1\rangle \vdash_n^{\text{PIL}} t : A$ (par inversibilité). Donc, par affaiblissement, $\Gamma \vdash_n^{\text{PIL}} t : A$. De plus, $\langle\Gamma_1\rangle \vdash_n^{\text{PIL}} \langle t\rangle : !A$. Donc $\langle\Gamma_1\rangle \vdash_n^{\text{PIL}} \langle\langle t\rangle\rangle : !!A$ (règle Promotion), puis par affaiblissement à nouveau, $\Gamma \vdash_n^{\text{PIL}} \langle\langle t\rangle\rangle : !!A$. □

Le Lemme de Substitution est toujours valide :

Lemme 9.1.4 (Substitution)**.** *Si* $\Gamma \vdash_n^{\text{PIL}} t_1 : A$ *et* $\Gamma, x : A \vdash_n^{\text{PIL}} t_2 : B$, *alors* $\Gamma \vdash_n^{\text{PIL}} t_2[x/t_1] : B$. *Autrement dit, la règle*

$$\frac{\Gamma \vdash_n^{\text{PIL}} t_1 : A \qquad \Gamma, x : A \vdash_n^{\text{PIL}} t_2 : B}{\Gamma \vdash_n^{\text{PIL}} t_2[x/t_1] : B}\ \text{Cut}$$

est admissible.

Démonstration. La preuve est très similaire à celle de SIL_n. Le seul cas différent est celui où $t_2 = \langle t\rangle$. Dans ce cas, $t_2[x/t_1] = \langle t[x/t_1]\rangle$. De plus $\Gamma, x : A \vdash_n^{\text{PIL}} \langle t\rangle : B$. Donc par la propriété d'inversibilité (9.1.1), la dernière règle de la preuve de $\Gamma, x : A \vdash_n^{\text{PIL}} t_2 : B$ est Promotion. Donc $B = !B'$, $\Gamma = \Gamma', \langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n$, Γ' ne comporte pas de variables $\langle y\rangle$ et $\langle x_i\rangle : !A_i \vdash_n^{\text{PIL}} t : B'$. Donc par affaiblissement (Propriété 9.1.2), $\langle x_i\rangle : !A_i, x : A \vdash_n^{\text{PIL}} t : B'$. Par hypothèse d'induction, $\langle x_i\rangle : !A_i \vdash_n^{\text{PIL}} t[x/t_1] : B'$, donc $\Gamma', \langle x_i\rangle : !A_i \vdash_n^{\text{PIL}} \langle t\rangle[x/t] : !B'$ en utilisant la règle Promotion. □

Lemme 9.1.5 (Méta Substitution)**.** *Si* $\Gamma \vdash_n^{\text{PIL}} \langle t_1\rangle : !A$ *et* $\Gamma, \langle x\rangle : !A \vdash_n^{\text{PIL}} t_2 : B$ *alors* $\Gamma \vdash_n^{\text{PIL}} t_2[x/t_1] : B$. *Autrement dit, la règle*

$$\frac{\Gamma \vdash_n^{\text{PIL}} \langle t_1\rangle : !A \qquad \Gamma, \langle x\rangle : !A \vdash_n^{\text{PIL}} t_2 : B}{\Gamma \vdash_n^{\text{PIL}} t_2[x/t_1] : B}\ \text{MCut}$$

est admissible.

Démonstration. On fait uniquement le cas où $t_2 = \langle t\rangle$. On sait que $\Gamma \vdash_n^{\text{PIL}} \langle t_1\rangle : !A$ implique, par inversibilité qu'il existe $\Delta = x_1 : A_1, \cdots, x_n : A_n$ tel que $\langle\Delta\rangle \vdash_n^{\text{PIL}} t_1 : A$ et $\Gamma = \Gamma', \langle\Delta\rangle$ et Γ' ne contient pas $\langle y\rangle$.

Si $t_2 = \langle t\rangle$, alors par propriété d'inversibilité la dernière règle de la preuve de $\Gamma, \langle x\rangle : !A \vdash_n^{\text{PIL}} \langle t\rangle : !B$ est Promotion. On doit prouver que $\Gamma \vdash_n^{\text{PIL}} \langle t[x/t_1]\rangle : !B$. On sait que $\langle\Delta\rangle \vdash_n^{\text{PIL}} t_1 : A$. Donc $\langle\Delta\rangle \vdash_n^{\text{PIL}} \langle t_1\rangle : !A$ par application de Promotion. De plus, on sait que $\Gamma, \langle x\rangle : !A \vdash_n^{\text{PIL}} \langle t\rangle : !B$ implique que $\langle\Delta\rangle, \langle x\rangle : !A \vdash_n^{\text{PIL}} t : B$. Par hypothèse d'induction, $\langle\Delta\rangle \vdash_n^{\text{PIL}} t[x/t_1] : B$. Donc, par Promotion puis Weakening $\Gamma \vdash_n^{\text{PIL}} \langle t[x/t_1]\rangle : !B$. □

9.2 Correction du système de types

Le système de types est toujours cohérent avec la définition de variables libres de Λ_R.

Propriété 9.2.1. *If* $\Gamma \vdash_n^{\text{PIL}} t : A$, *then*

- $x \in \text{FV}^{\mathsf{let}}(t) \Rightarrow \langle x \rangle : !B \in \Gamma$
- $\text{FV}(t) \subset \text{dom}(\Gamma)$

Démonstration. Par induction sur t. □

Lemme 9.2.2 (Réduction en tête)**.** *Si* $t \to_{\text{head}} t'$ *et* $\Gamma \vdash_n^{\text{PIL}} t : A$, *alors* $\Gamma \vdash_n^{\text{PIL}} t' : A$

Démonstration. Même démonstration que pour le Lemme 8.2.2. Par analyse des cas de t.

- Si $t = (\lambda x.t_2)t_1$, alors $t' = t_2[x/t_1]$. Si $\Gamma \vdash_n^{\text{PIL}} (\lambda x.t_2)t_1 : A$, par inversibilité, $\Gamma, x : B \vdash_n^{\text{PIL}} t_2 : A$ et $\Gamma \vdash_n^{\text{PIL}} t_1 : B$. Par le Lemme de Substitution 9.1.4, $\Gamma \vdash_n^{\text{PIL}} t' : A$.
- Si $t = \mathsf{let}\ \langle x \rangle = \langle t_1 \rangle\ \mathsf{in}\ t_2$, alors $t' = t_2[x/t_1]$. Si $\Gamma \vdash_n^{\text{PIL}} \mathsf{let}\ \langle x \rangle = \langle t_1 \rangle\ \mathsf{in}\ t_2 : A$, par inversibilité, alors $\Gamma, \langle x \rangle : !B \vdash_n^{\text{PIL}} t_2 : A$ et $\Gamma \vdash_n^{\text{PIL}} \langle t_1 \rangle : !B$. Par le deuxième Lemme de Substitution 9.1.5, $\Gamma \vdash_n^{\text{PIL}} t' : A$

□

Lemme 9.2.3 (Compositionalité)**.** *Si* $\Gamma \vdash_n^{\text{PIL}} E[t] : A$, *il existe* $\Gamma' \supseteq \Gamma$ *tel que*

- $\Gamma' \vdash_n^{\text{PIL}} t : B$,
- *pour tout* t', $\Gamma' \vdash_n^{\text{PIL}} t' : B \Rightarrow \Gamma \vdash_n^{\text{PIL}} E[t'] : A$

Démonstration. Même démonstration que pour le Lemme 8.2.3. Par induction sur E.

- Si $E = []$, alors $\Gamma' \stackrel{\text{def}}{=} \Gamma$ et $B \stackrel{\text{def}}{=} A$ fonctionne.
- Si $E = E'\ t_1$ (resp. $E = t_1\ E'$), alors par inversibilité, $\Gamma \vdash_n^{\text{PIL}} E'[t] : C \to A$ et $\Gamma \vdash_n^{\text{PIL}} t_1 : C$ (resp. $\Gamma \vdash_n^{\text{PIL}} E'[t] : C$ et $\Gamma \vdash_n^{\text{PIL}} t_1 : C \to A$). Par hypothèse d'induction, il existe $\Gamma' \supseteq \Gamma$ et B tel que $\Gamma' \vdash_n^{\text{PIL}} t : B$. De plus, pour tout t' tel que $\Gamma' \vdash_n^{\text{PIL}} t' : B$, $\Gamma \vdash_n^{\text{PIL}} E'[t'] : C \to A$ (resp. $\Gamma \vdash_n^{\text{PIL}} E'[t'] : C$). Donc $\Gamma \vdash_n^{\text{PIL}} E[t'] : A$ par la règle Application.
- Si $E = \lambda x.E'$, alors, par inversibilité, $A = C \to D$ et $\Gamma, x : C \vdash_n^{\text{PIL}} E'[t] : D$. Par hypothèse d'induction, il existe B et $\Gamma' \supseteq \Gamma$ tel que $\Gamma', x : C \vdash_n^{\text{PIL}} t : B$. De plus, pour tout t' tel que $\Gamma', x : C \vdash_n^{\text{PIL}} t' : B$, $\Gamma, x : C \vdash_n^{\text{PIL}} E'[t'] : D$. Donc $\Gamma \vdash_n^{\text{PIL}} E[t'] : A$ par la règle Abstraction.
- Si $E = \mathsf{let}\ \langle x \rangle = t_1\ \mathsf{in}\ E'$ (resp. $E = \mathsf{let}\ \langle x \rangle = E'\ \mathsf{in}\ t_1$), alors, par inversibilité, il existe C tel que $\Gamma, \langle x \rangle : !C \vdash_n^{\text{PIL}} E'[t] : A$ et $\Gamma \vdash_n^{\text{PIL}} t_1 : !C$ (resp. $\Gamma, \langle x \rangle : !C \vdash_n^{\text{PIL}} t_1 : A$ et $\Gamma \vdash_n^{\text{PIL}} E'[t] : !C$). Par hypothèse d'induction, il existe B et $\Gamma' \supseteq \Gamma$ tels que $\Gamma', \langle x \rangle : !C \vdash_n^{\text{PIL}} t : B$ (resp. $\Gamma' \vdash_n^{\text{PIL}} t : B$). De plus, pour tout t' tel que $\Gamma', x : C \vdash_n^{\text{PIL}} t' : B$ (resp. $\Gamma' \vdash_n^{\text{PIL}} t' : B$), $\Gamma, x : C \vdash_n^{\text{PIL}} E'[t'] : A$ (resp. $\Gamma \vdash_n^{\text{PIL}} E'[t'] : !C$). Alors $\Gamma \vdash_n^{\text{PIL}} E[t'] : A$ par la règle Let.

□

Finalement, on obtient le résultat de correction du système.

Théorème 9.2.4 (Réduction du sujet)**.** *Si* $\Gamma \vdash_n^{\text{PIL}} t : A$ *et* $t \to t'$ *alors* $\Gamma \vdash_n^{\text{PIL}} t' : A$.

Démonstration. Même démonstration que pour le Théorème 8.2.5. Si $t \to t'$, il existe E, s, s' tel que $t = E[s]$, $s \to_{\text{head}} s'$ et $t' = E[s']$. Par le Lemme de Composition, $\Gamma' \vdash_n^{\text{PIL}} s : B$. Par le Lemme 9.2.2, $\Gamma' \vdash_n^{\text{PIL}} s' : B$. Par compositionnalité à nouveau, $\Gamma \vdash_n^{\text{PIL}} E[s'] = t' : A$. □

Remarque 9.2.5. De la même manière que pour SIL les résultats de compositionnalité (Lemme 9.2.3), d'affaiblissement (Propriété 9.1.2), de substitution (Lemmes 9.1.4 et 9.1.5) et de réduction du sujet (Théorème 9.2.4) sont toujours valides dans *tout sous-système de* PIL_n. ■

Théorème 9.2.6 (Normalisation forte)**.** *Si* $\Gamma \vdash_n^{\text{PIL}} t : A$ *alors tout chemin de réduction de* t *est fini.*

Démonstration. Même démonstration que pour le Théorème 8.2.8 (exactement). En oubliant les notations nouvelles (modalité !, termes gelés $\langle t \rangle$ et lieurs let) on obtient le λ-calcul simplement typé, et toute réduction dans notre langage correspond à au moins une réduction en λ-calcul (Propriété 7.5.13). Comme le λ-calcul simplement typé est fortement normalisable, il en va de même pour notre langage. □

9.3 Typage en calcul des séquents

Dans cette section on donne une présentation sous forme de calcul des séquents de notre système de types. Ce système est appelé PIL_s. Les règles de PIL_s sont présentées dans la Figure 9.2.

9.4 Équivalence des deux systèmes

Comme pour SIL_n et SIL_s, PIL_n et PIL_s sont équivalents.

Théorème 9.4.1. *Le jugement* $\Gamma \vdash_n^{\text{PIL}} t : A$ *est dérivable si et seulement si* $\Gamma \vdash_s^{\text{PIL}} t : A$ *est dérivable.*

Démonstration. De la même manière que la preuve du Théorème 8.4.1, on définit des traductions $\mathcal{N}$ et $\mathcal{S}$ passant de PIL_s à PIL_n, et inversement. Dans cette partie, on ne donne les traductions que pour la règle de promotion.

La preuve en calcul des séquents

$$\textsc{Promotion}\ \frac{\begin{matrix}\pi\\ \langle x_i \rangle : !A_i \vdash_s^{\text{PIL}} t : B\end{matrix}}{\langle x_i \rangle : !A_i \vdash_s^{\text{PIL}} \langle t \rangle : !B}$$

est changée en *pseudo* déduction naturelle en

$$\textsc{Promotion}\ \frac{\begin{matrix}\mathcal{N}(\pi)\\ \langle x_i \rangle : !A_i \vdash_n^{\text{PIL}} t : B\end{matrix}}{\langle x_i \rangle : !A_i \vdash_n^{\text{PIL}} \langle t \rangle : !B}$$

$$\textsc{Let}\ \frac{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{PIL}} t : B}{\Gamma, y : !A \vdash_s^{\mathrm{PIL}} \mathsf{let}\ \langle x\rangle = y\ \mathsf{in}\ t : B} \qquad \textsc{Axiom}\ \frac{}{x : A \vdash_s^{\mathrm{PIL}} x : A}$$

$$\textsc{Weak}\ \frac{\Gamma \vdash_s^{\mathrm{PIL}} t : B}{\Gamma, x : A \vdash_s^{\mathrm{PIL}} t : B} \qquad \textsc{MetaWeak}\ \frac{\Gamma \vdash_s^{\mathrm{PIL}} t : B}{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{PIL}} t : B}$$

$$\textsc{Cont}\ \frac{\Gamma, y : A, z : A \vdash_s^{\mathrm{PIL}} t : C}{\Gamma, x : A \vdash_s^{\mathrm{PIL}} t[y/x][z/x] : C} \qquad \textsc{MetaCont}\ \frac{\Gamma, \langle y\rangle : !A, \langle z\rangle : !A \vdash_s^{\mathrm{PIL}} t : C}{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{PIL}} t[y/x][z/x] : C}$$

$$\textsc{}{\to}\textsc{Left}\ \frac{\Gamma, x : B \vdash_s^{\mathrm{PIL}} t : C \qquad \Delta \vdash_s^{\mathrm{PIL}} t' : A}{\Gamma, \Delta, f : A \to B \vdash_s^{\mathrm{PIL}} t[x/f\ t'] : C} \qquad \textsc{}{\to}\textsc{Right}\ \frac{\Gamma, x : A \vdash_s^{\mathrm{PIL}} t : B}{\Gamma \vdash_s^{\mathrm{PIL}} \lambda x.t : A \to B}$$

$$\textsc{Promotion}\ \frac{\langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n \vdash_s^{\mathrm{PIL}} t : B}{\langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n \vdash_s^{\mathrm{PIL}} \langle t\rangle : !B} \qquad \textsc{Dereliction}\ \frac{\Gamma, x : A \vdash_s^{\mathrm{PIL}} t : B}{\Gamma, \langle x\rangle : !A \vdash_s^{\mathrm{PIL}} t : B}$$

$$\textsc{Cut}\ \frac{\Gamma \vdash_s^{\mathrm{PIL}} t' : A \quad \Delta, x : A \vdash_s^{\mathrm{PIL}} t : B}{\Gamma, \Delta \vdash_s^{\mathrm{PIL}} t[x/t'] : B}$$

FIGURE 9.2 – PIL_s rules

La preuve en *pseudo* déduction naturelle

$$\textsc{Promotion}\ \frac{\begin{array}{c}\pi\\ \langle x_i\rangle : !A_i \vdash_n^{\text{PIL}} t : B\end{array}}{\Gamma, \langle x_i\rangle : !A_i \vdash_n^{\text{PIL}} \langle t\rangle : !B}$$

est changée en calcul des séquents en

$$\begin{array}{l}\textsc{Promotion}\\ \textsc{Weakening}\end{array}\ \frac{\begin{array}{c}\mathcal{S}(\pi)\\ \langle x_i\rangle : !A_i \vdash_s^{\text{PIL}} t : B\end{array}}{\begin{array}{c}\langle x_i\rangle : !A_i \vdash_s^{\text{PIL}} \langle t\rangle : !B\\ \hline \Gamma, \langle x_i\rangle : !A_i \vdash_s^{\text{PIL}} \langle t\rangle : !B\end{array}}$$

□

Dans ce qui suit, on dira que $\Gamma \vdash^{\text{PIL}} t : A$ dans PIL lorsque $\Gamma \vdash_n^{\text{PIL}} t : A$ PIL_n (ou, de manière équivalente, lorsque $\Gamma \vdash_s^{\text{PIL}} t : A$ in PIL_s).

9.5 Encodage de la Logique Intuitionniste

Il est connu depuis [Gir87] et la traduction de Girard qu'il est possible d'encoder la Logique Intuitionniste dans la Logique Linéaire. On utilise l'*implication intuitionniste* $A \Rightarrow B \stackrel{\text{def}}{=} !A \to B$. La traduction A^g est donnée par

$$\begin{array}{rcl}\alpha^g & \stackrel{\text{def}}{=} & \alpha\\ (A \to B)^g & \stackrel{\text{def}}{=} & A^g \Rightarrow B^g\end{array}$$

De même un terme en λ-calcul peut être traduit en terme de Λ_R

$$\begin{array}{rcll} x^g & = & x & \\ \lambda x.t^g & = & \lambda X.\,\mathsf{let}\,\langle x\rangle = X\,\mathsf{in}\,t^g & X \notin \text{FV}(t)\\ (t_1\ t_2)^g & = & t_1^g\ \langle t_2^g\rangle & \end{array}$$

On a alors le théorème suivant

Théorème 9.5.1. *Le jugement $x_1 : A_1, \cdots, x_n : A_n \vdash t : A$ est valide (en Logique Intuitionniste) si et seulement si $\langle x_1\rangle : !A_1^g, \cdots, \langle x_n\rangle : !A_n^g \vdash^{\text{PIL}} t^g : A^g$ est valide dans* PIL.

Démonstration. On effectue la preuve dans les versions en pseudo déduction naturelle de la Logique Intuitionniste et de PIL pour profiter de l'inversibilité. La preuve de l'équivalence peut se faire par induction sur t.

Si $\Gamma = x_1 : A_1, \cdots, x_n : A_n$, posons $\langle\Gamma^g\rangle = \langle x_1\rangle : !A_1^g, \cdots, \langle x_n\rangle : !A_n^g$.

— Si $t = x$, alors

$$\begin{array}{rl} & \Gamma \vdash_n x : A\\ \iff & x : A \in \Gamma\\ \iff & \langle x\rangle : !A^g \in \langle\Gamma^g\rangle\\ \iff & \langle\Gamma^g\rangle \vdash_n^{\text{PIL}} x : A^g\end{array}$$

— Si $t = t_1\ t_2$, alors

$$\begin{array}{rl} & \Gamma \vdash_n t_1\ t_2 : A \\ \iff & \Gamma \vdash_n t_1 : B \to A \wedge \Gamma \vdash_n t_2 : B \\ \iff & \langle \Gamma^g \rangle \vdash_n^{\mathrm{PIL}} t_1^g : B^g \Rightarrow A^g \wedge \langle \Gamma^g \rangle \vdash_n^{\mathrm{PIL}} t_2^g : B^g \\ \iff & \langle \Gamma^g \rangle \vdash_n^{\mathrm{PIL}} t_1^g : B^g \Rightarrow A^g \wedge \langle \Gamma^g \rangle \vdash_n^{\mathrm{PIL}} \langle t_2^g \rangle : !B^g \quad (*) \\ \iff & \langle \Gamma^g \rangle \vdash_n^{\mathrm{PIL}} t_1^g\ \langle t_2^g \rangle : A^g \end{array}$$

($*$) L'existence de B (dans le sens de bas en haut) est due au Lemme 9.5.2.

— Si $t = \lambda x.t'$, alors

$$\begin{array}{rll} & \Gamma \vdash_n \lambda x.t' : A \to B & \\ \iff & \Gamma, x : A \vdash_n t' : B & \\ \iff & \langle \Gamma^g \rangle, \langle x \rangle : !A^g \vdash_n^{\mathrm{PIL}} t'^g : B^g & \\ \iff & \langle \Gamma^g \rangle, \langle x \rangle : !A^g, X : !A^g \vdash_n^{\mathrm{PIL}} t'^g : B^g & (*) \\ \iff & \langle \Gamma^g \rangle \vdash_n^{\mathrm{PIL}} \lambda X.\, \mathsf{let}\ \langle x \rangle = X\ \mathsf{in}\ t'^g : A^g \Rightarrow B^g & \end{array}$$

($*$) Le sens de bas en haut est assuré par le fait que $X \notin \mathrm{FV}(t')$.

□

Lemme 9.5.2. *Si* $\Gamma \vdash_n^{\mathrm{PIL}} t^g : A$, *alors* $\langle \Gamma^{*g} \rangle \vdash_n^{\mathrm{PIL}} t^g : A^{*g}$, *où* A^* *est obtenu en supprimant tous les* ! *de* A, *et où* Γ^* *est obtenu en changeant tous les* $x : A$ *en* $x : A^*$ *et les* $\langle x \rangle : !A$ *en* $x : A^*$.

Démonstration. Par induction sur t. On suppose que $\Gamma = \Delta_1, \langle \Delta_2 \rangle$ où Δ_1 et Δ_2 ne contiennent pas de variables de la forme $\langle x \rangle$.

— Si $t = x$, alors $x : A \in \Gamma$ ou $\langle x \rangle : !A \in \Gamma$. Dans les deux cas, $\langle x \rangle : !A^{*g} \in \langle \Gamma^{*g} \rangle$. Donc $\langle \Gamma^{*g} \rangle \vdash_n^{\mathrm{PIL}} x : A^{*g}$.

— Si $t = \lambda x.t'$, alors $\Gamma, \langle x \rangle : !A \vdash_n^{\mathrm{PIL}} t'^g : B$. Donc $\langle \Gamma^{*g} \rangle, \langle x \rangle : !A^{*g} \vdash_n^{\mathrm{PIL}} t'^g : B^{*g}$. Donc $\langle \Gamma^{*g} \rangle \vdash_n^{\mathrm{PIL}} (\lambda x.t')^g : A^{*g} \Rightarrow B^{*g}$.

— Si $t = t_1\ t_2$, alors $\Gamma \vdash_n^{\mathrm{PIL}} t_1^g : !A \to B$ et $\langle \Delta_2 \rangle \vdash_n^{\mathrm{PIL}} t_2^g : A$. Donc $\langle \Gamma^{*g} \rangle \vdash_n^{\mathrm{PIL}} t_1^g : !A^{*g} \to B^{*g}$ et $\langle \Delta_2^{*g} \rangle \vdash_n^{\mathrm{PIL}} t_2^g : A^{*g}$. Donc $\langle \Gamma^{*g} \rangle \vdash_n^{\mathrm{PIL}} t_1\ t_2^g : B^{*g}$.

□

Ce théorème n'est pas vrai dans SIL : la preuve échoue au moment où elle utilise la règle Promotion. L'impossibilité est même plus forte que cela : le jugement $\alpha, \alpha \to \beta, \beta \to \gamma \vdash \gamma$ (sans terme de preuve) est dérivable en Logique Intuitionniste (les types α, β et γ sont atomiques), mais il n'existe pas de terme t tel que le jugement $\langle a \rangle : !\alpha, \langle f \rangle : !(\alpha \Rightarrow \beta), \langle b \rangle : !(\beta \Rightarrow \gamma) \vdash^{\mathrm{SIL}} t : \gamma$ soit dérivable.

Remarque 9.5.3. On a montré qu'une formule A de la Logique Intuitionniste était prouvable en Logique Intuitionniste si et seulement si A^g était prouvable dans PIL. On notera toutefois que PIL contient déjà la Logique Intuitionniste : la formule A est aussi prouvable dans PIL.

Le point intéressant de notre théorème est qu'il utilise un encodage qui *met en évidence le rôle de programme et de donnée des termes dans le calcul*. En effet, un terme de type

$A \to B$ en Logique Intuitionniste est traduit en un terme de type $A \Rightarrow B = !A \to B$. Le terme est donc un programme (type $\to$) prenant en paramètre une donnée (type !). Cette traduction est également en lien avec la construction de Kleisli sur la comonade !. ■

Plus tard dans cette thèse (Section 11.4.3) on montrera qu'il est possible d'encoder la Logique Classique dans un langage dérivé de Λ_R avec cette même traduction.

9.6 Staged computation

Dans cette section on donne les interprétations logiques des systèmes de calcul présentés dans la Section 7.5.

9.6.1 Logique temporelle

Davis dans [Dav96] et Taha et Sheard dans [TS97] donnent une implémentation de staged computation dans le cadre de la logique temporelle. Si $\langle t \rangle$ est un terme de type $\bigcirc A$ s'exécutant au niveau n dans l'environnement Γ^n, alors t est de type A et s'exécute au niveau $n+1$ dans l'environnement Γ^{n+1}, où Γ^n et Γ^{n+1} n'ayant rien à voir l'un avec l'autre.

On peut typer un sous-ensemble de termes de Λ_T en utilisant la logique modale. L'ensemble des types A est donné par

$$A \stackrel{\text{def}}{=} \alpha \mid A \to B \mid \bigcirc A$$

Le type $\bigcirc A$ est le type des termes de type A s'exécutant au niveau suivant.

Un *contexte de typage* $\Gamma = x_1 : A_1^{p_1}, \cdots, x_n : A_n^{p_n}$ est un ensemble d'hypothèses où toutes les variables x_i sont différentes et où chaque $p_i \in \mathbb{N}$. Ces entiers correspondent aux niveaux des variables x_i dans le terme à typer. Comme on l'a noté dans la Section 7.5.2.12, à chaque variable x_i libre d'un terme est associé un unique niveau p_i.

Le système de types est donné par les règles suivantes. On dira qu'un terme $t \in \Lambda_T$ est typable s'il existe un type A et un environnement Γ tels qu'il existe une dérivation dans ce système telle que $\Gamma \vdash^n t : A$. L'entier n désigne le niveau d'exécution du terme.

9.6.1.1 Fragment simplement typé

Chaque niveau n a son propre environnement de typage, ce qui est vérifié par la règle de l'axiome et de l'abstraction.

$$\text{Var}\ \frac{}{\Gamma, x : A^n \vdash^n x : A} \qquad \text{Abstraction}\ \frac{\Gamma, x : A^n \vdash^n t : B}{\Gamma \vdash^n \lambda x.t : A \to B} \qquad \text{Application}\ \frac{\Gamma \vdash^n t : A \to B \qquad \Gamma \vdash^n t' : A}{\Gamma \vdash^n t\ t' : B}$$

9.6.1.2 Fragment temporel

Le terme $\langle t \rangle$ est bien typé au niveau n si t l'est au niveau $n+1$ Un niveau peut faire référence au niveau inférieur en utilisant l'échappement $\sim$

$$\frac{\Gamma \vdash^{n+1} t : A}{\Gamma \vdash^{n} \langle t \rangle : \bigcirc A}\ \textsc{Next} \qquad \frac{\Gamma \vdash^{n} t : \bigcirc A}{\Gamma \vdash^{n+1} \sim t : A}\ \textsc{Prev}$$

Théorème 9.6.1 (Correction du système de types [Dav96]). *Si* $t \overset{\text{CBV}}{\to} t'$ *et* $\Gamma \vdash^n t : A$, *alors* $\Gamma \vdash t' : A$

On a vu dans la Section 7.5 que la réduction des termes imposait une gestion rigoureuse des niveaux d'exécution, car les sous-termes dont on forçait la réduction (avec le caractère d'échappement $\sim$) pouvaient ne pas être clos. Il faut donc s'assurer que les variables libres des sous-termes que l'on souhaite réduire soient d'un niveau supérieur afin que leur exécution soit retardée.

Cette gestion très fine des niveaux impose l'indexation des jugements de typage par le niveau d'exécution du terme. Il est par conséquent impossible en l'état de partager un même terme retardé entre plusieurs niveaux, ce qui risquerait de casser la correction du système de types. Par exemple, il n'existe pas d'opérateur permettant l'exécution d'un terme, c'est à dire son passage du niveau 1 au niveau 0. Une autre façon de le voir est le fait qu'il n'existe pas de terme de type $\bigcirc A \to A$.

9.6.2 Logique modale

Comme décrit dans [BP01 ; DP96], les types sont de la forme

$$A \overset{\text{def}}{=} \alpha \mid A \to B \ \mid\ \Box A$$

où α désigne n'importe quel type de base.

Un *contexte de typage* $\Gamma = x_1 : A_1, \cdots, x_n : A_n$ est un ensemble d'hypothèses où toutes les variables x_i sont différentes. Le *domaine* de Γ est l'ensemble $\{x \mid \exists A, x : A \in \Gamma\}$.

Le système de types est donné par les règles suivantes. On dira qu'un terme $t \in \Lambda_M$ est typable s'il existe une type A et deux environnements Γ, Δ de domaines disjoints tels qu'il existe une dérivation dans ce système telle que $\Delta; \Gamma \vdash_\Box t : A$.

Fragment simplement typé

$$\frac{}{\Delta; \Gamma, x : A \vdash_\Box x : A}\ \textsc{Var} \qquad \frac{\Delta; \Gamma, x : A \vdash_\Box t : B}{\Delta; \Gamma \vdash_\Box \lambda x.t : A \to B}\ \textsc{Abstraction} \qquad \frac{\Delta; \Gamma \vdash_\Box t : A \to B \qquad \Gamma \vdash_\Box t' : A}{\Delta; \Gamma \vdash_\Box t\ t' : B}\ \textsc{Application}$$

Fragment modal

$$\textsc{Box}\ \frac{\Delta;\vdash_{\Box} t : B}{\Delta;\Gamma : !A_n \vdash_{\Box} \langle t\rangle : !B} \qquad \textsc{Let}\ \frac{\Delta;\Gamma, x : \Box A \vdash_{\Box} t' : !A \qquad \Delta, x : A;\Gamma \vdash_{\Box} t : B}{\Delta;\Gamma \vdash_{\Box} \mathsf{let}\ \langle x\rangle = t'\ \mathsf{in}\ t : B}$$

$$\textsc{MVar}\ \frac{}{\Delta, x : A;\Gamma \vdash_{\Box} x : A}$$

Contrairement au cas de la logique temporelle, le système en logique modale s'attache à retarder l'exécution de termes clos. Il est ainsi plus facile de faire passer un niveau retardé à un autre niveau. Si $\Box A$ est le type des termes dont l'exécution est retardée, on dispose par exemple de la règle $\Box A \to A$ qui fait passer une terme retardé du niveau $n+1$ au niveau n, et de la règle $\Box A \to \Box\Box A$ qui fait passer un terme retardé du niveau $n+1$ au niveau $n+2$.

9.6.2.1 Logique modale et logique linéaire

L'implémentation en logique modale utilise la modalité $\Box$ de la même manière qu'on utilise !.

Propriété 9.6.2. *Le jugement $\Delta;\Gamma \vdash_{\Box} t : A$ est dérivable en logique modale si et seulement si le jugement $\widetilde{\Delta}, \underline{\Gamma} \vdash^{\mathrm{PIL}} t : \underline{A}$ l'est en dans* PIL*, où $\underline{A}$ et $\underline{\Gamma}$ sont obtenus à partir de A et Γ en remplaçant tous les signes $\Box$ par ! et où $\widetilde{\Delta} \stackrel{def}{=} \{\langle x\rangle : !A \mid x : A \in \Delta\}$.*

Démonstration. Par induction sur la dérivation de $\Delta;\Gamma \vdash_{\Box} t : A$. □

Ce résultat n'est pas surprenant, étant donné qu'il est connu au moins depuis le travail de Schellinx dans [Sch96] et celui de Martini et Masini dans [MM94] que la logique modale S4 peut être traduite en logique linéaire en oubliant la gestion des ressources.

9.7 Conclusion

On a donné un nouveau système logique pour Λ_R. Celui ci s'inspire plus fortement de la Logique Linéaire. Là aussi on a donné une version en calcul des séquents et une version en pseudo déduction naturelle. Il a les mêmes propriétés de correction que SIL (réduction du sujet, normalisation forte, ...).

C'est un système strictement plus grand que SIL. Contrairement à SIL, il permet d'encoder toute la Logique Intuitionniste. De plus, la traduction utilisée, fait apparaitre clairement le rôle de programme et de donnée dans un terme.

On va détailler un peu plus dans le chapitre suivant les différences entre SIL et PIL.

Chapitre 10

Comparaison entre PIL et SIL

10.1 Usage de la déréliction

La règle Dereliction peut avoir deux rôles : d'une part *permettre la substitution dans les termes gelés* $\langle t \rangle$, et d'autre part *exécuter des termes gelés*. Pour montrer ces résultats, on va observer ce qui se passe dans les systèmes PIL et SIL lorsque la règle Dereliction est supprimée. Les systèmes obtenus sont notés PIL$'$ et SIL$'$.

Remarque 10.1.1. Les système PIL$'$ et SIL$'$ peuvent être obtenus en supprimant la règle Dereliction des systèmes en déduction naturelle ou en calcul des séquents. Ces deux solutions sont équivalentes : en effet, les preuves d'équivalence de $\text{PIL}_n/\text{PIL}_s$ et de $\text{SIL}_n/\text{SIL}_s$ utilisent les transformations $\mathcal{N}$ et $\mathcal{S}$ qui font apparaître la règle Dereliction uniquement quand la preuve initiale utilise *déjà* la règle Dereliction. ■

10.1.1 Substitution dans les termes gelés

Dans PIL, le résultat principal est le suivant

Propriété 10.1.2. *Si* $\Gamma, \langle x \rangle : !A \vdash^{\text{PIL}'} t : B$ *dans* PIL$'$*, alors* $x \notin \text{FV}(t)$.

Démonstration. Par induction sur t. Étant donné que la règle Dereliction est interdite, le cas $\Gamma, \langle x \rangle : !A \vdash^{\text{PIL}'} x : A$ est impossible. □

Dans SIL, le résultat est plus faible, car le $\langle x \rangle : !A \vdash \langle x \rangle : !A$ est typable dans SIL$'$ mais pas dans PIL$'$.

Propriété 10.1.3. *Si* $\Gamma, \langle x \rangle : !A \vdash^{\text{SIL}'} t : B$ *dans* SIL$'$*, alors* $x \notin \text{FV}^{\lambda}(t)$.

Démonstration. Par induction sur t. □

Remarque 10.1.4. Il reste possible dans SIL$'$ d'effectuer des substitutions dans les termes gelés. Par exemple, le terme $\mathsf{let}\ \langle x \rangle = \langle \lambda y.y \rangle\ \mathsf{in}\ \langle x \rangle$ est bien typé dans SIL$'$ et se réduit en $\langle \lambda y.y \rangle$. ■

10.1.2 Relation entre les deux systèmes de types

La seule différence entre PIL et SIL est la règle SoftPromotion. Il est clair que chaque dérivation dans SIL est encore valide dans PIL. Le système SIL est même strictement plus petit que PIL : le jugement $y : !A \vdash \mathsf{let}\ \langle x\rangle = y\ \mathsf{in}\ \langle\langle x\rangle\rangle : !!A$ est dérivable dans PIL alors qu'il ne l'est pas dans SIL[6].

$$\text{SIL} \subsetneq \text{PIL}$$

On détaillera dans la Section 10.2 les preuves manquantes de SIL.

L'intérêt du système SIL est dans l'utilisation de sa règle Dereliction.

Propriété 10.1.5. *Le système* SIL′ *contient le système* PIL′.

$$\text{PIL}' \subsetneq \text{SIL}'$$

Démonstration. Par induction sur la forme de la preuve dans PIL′. Le seul cas non trivial concerne la règle Promotion. Si $\langle\Gamma\rangle \vdash^{\text{PIL}'} \langle t\rangle : !A$ dans PIL′, alors la Propriété 10.1.2 affirme que t est clos. Alors $\vdash^{\text{PIL}'} \langle t\rangle : !A$ dans PIL′ (la forme de la preuve n'est pas changée, on a juste supprimé l'environnement de typage) et donc $\vdash^{\text{PIL}'} t : A$. Par hypothèse d'induction, $\vdash^{\text{SIL}'} t : A$ est dérivable dans SIL′, puis $\Gamma \vdash^{\text{SIL}'} t : A$ (par affaiblissement) et finalement, $\langle\Gamma\rangle \vdash^{\text{SIL}'} \langle t\rangle : !A$. □

La règle SoftPromotion embarque avec elle tout ce qui est nécessaire pour permettre la substitution dans les termes gelés. Par exemple, le terme $\lambda x.\, \mathsf{let}\ \langle y\rangle = x\ \mathsf{in}\ \langle y\rangle$ est bien typé dans SIL′ alors qu'il ne l'est pas dans PIL′.

10.1.3 Exécution de termes gelés

Les deux systèmes jouissent de la propriété suivante

Propriété 10.1.6 (Non Interference)**.** *Si* $\Gamma, \langle x\rangle : !A \vdash t_1 : A$ *est dérivable sans déréliction (dans* PIL_s/PIL_n *ou* SIL_s/SIL_n*) et si* t_2 *est la forme normale de* t_1*, alors, pour tout terme* t*,* $t_2[x/t]$ *est la forme normale de* $t_1[x/t]$*.*

Démonstration. En fonction du système PIL ou SIL :

- Dans PIL, $x \notin \text{FV}(t_1)$, donc $\lambda x \notin \text{FV}(t_2)$. Donc $t_1[x/t] = t_1$ et $t_2[x/t] = t_2$.
- Dans SIL_s, comme $t_1 \xrightarrow{*} t_2$ et que $\Gamma, \langle x\rangle : !A \vdash^{\text{SIL}'} t_1 : B$, alors $\Gamma, \langle x\rangle : !A \vdash^{\text{SIL}'} t_2 : B$ (le Théorème de Réduction du Sujet est toujours valable dans SIL′, par la Remarque 8.2.7). Par la Propriété 10.1.3, $x \notin \text{FV}^{\lambda}(t_2)$. Donc soit x n'est pas libre dans t_2 or x est environnée par $\langle\rangle$ dans t_2. Dans les deux cas, $t_2[x/t]$ est en forme normale car t_2 l'est et qu'on ne réduit pas dans les $\langle\cdots\rangle$.

□

Remarque 10.1.7. Cette propriété utilise le fait que terme t dans $\mathsf{let}\ \langle x\rangle = \langle t\rangle\ \mathsf{in}\ \langle t'\rangle$ n'est jamais réduit. ■

6. La preuve de ce résultat est facilement exprimable en utilisant la version inversible de SIL.

Dans le système SIL, la déréliction correspond *exactement* à l'exécution d'un terme.

Propriété 10.1.8. *Si* $\Gamma, \langle x\rangle : !A \vdash_n^{\text{SIL}} t : B$ *est dérivable dans* SIL avec *une déréliction* Dereliction *sur* x*, alors* $x \in \text{FV}^\lambda(t)$.

Démonstration. Par induction sur t. Les cas intéressants sont lorsque t est une variable et t est gelé.

— Si $t : x$, alors t est typable par une déréliction et $x \in \text{FV}^\lambda(t)$.

— Le cas $t = \langle t'\rangle$ est impossible, alors, si $\Gamma = \langle x_1\rangle : !A_1, \cdots, x_n : !A_n, y_1 : B_1, \cdots, y_m : B_m$, alors $x_1 : A_1, \cdots, x_n : A_n, x : A \vdash_n^{\text{SIL}} t' : B'$ (où $B = !B'$). Donc on ne peut avoir de déréliction sur x.

□

Remarque 10.1.9. Cette propriété est fausse dans PIL car la dérivation du jugement $\Gamma, \langle x\rangle : !A \vdash^{\text{PIL}} \langle x\rangle : !A$ dans PIL utilise la déréliction sur x alors que $x \notin \text{FV}^\lambda(t)$. ■

10.2 Digging

Comme on l'a vu dans la Section 6.4, la règle de promotion (et donc toute l'expressivité de la Logique Linéaire) peut être retrouvée à partir des règles SoftPromotion et Digging [Laf02]. On va montrer comment modifier SIL en lui ajoutant la règle Digging pour retrouver l'expressivité de PIL.

Dans cette partie on utilise des motifs p définis inductivement par $p = x \mid \langle p\rangle$, et les environnements de typage sont de la forme $p_1 : A_1, \cdots, p_n : A_n$. Le système de types SIL_d est donné dans la Figure 10.1 Les règles SoftPromotion et Dereliction sont modifiées pour l'utilisation des motifs p. On a le résultat suivant :

Théorème 10.2.1 (Correction)**.** *Si* $\Gamma \vdash_d t : A$ *alors* $\Gamma' \vdash^{\text{PIL}} t : A$ *où* Γ' *est construit à partir de* Γ *où tous les motifs sont aplatis :* $\langle x\rangle_n : !^n A$ *devient* $\langle x\rangle : !A$ *et* $x : A$ *est inchangé.*

Démonstration. Par induction sur la preuve de $\Gamma \vdash_d t : A$. Les cas intéressants sont pour les règles Digging, Dereliction et SoftPromotion.

— Digging est juste l'identité dans PIL.

— Dereliction dans SIL_d, où $p = x$, ne change pas dans PIL. Autrement, c'est l'identité dans PIL.

— SoftPromotion dans SIL_d où $p = x$ est la règle Promotion composée avec la règle de Dereliction dans PIL. Sinon, c'est juste la règle Promotion.

□

La réciproque est également vraie.

Théorème 10.2.2 (Complétude)**.** *Si* $\Gamma \vdash^{\text{PIL}} t : A$*, alors* $\Gamma \vdash_d t : A$.

Démonstration. Ce résultat est prouvé en dérivant la règle Promotion de PIL dans SIL_d :

$$\textsc{Let}\ \frac{\Gamma, \langle x\rangle : !A \vdash_d t : B}{\Gamma, y : !A \vdash_d \mathsf{let}\ \langle x\rangle = y\ \mathsf{in}\ t : B} \qquad \textsc{Axiom}\ \frac{}{x : A \vdash_d x : A}$$

$$\textsc{Weak}\ \frac{\Gamma \vdash_d t : B}{\Gamma, x : A \vdash_d t : B} \qquad \textsc{MetaWeak}\ \frac{\Gamma \vdash_d t : B}{\Gamma, \langle x\rangle : !A \vdash_d t : B}$$

$$\textsc{Cont}\ \frac{\Gamma, y : A, z : A \vdash_d t : C}{\Gamma, x : A \vdash_d t[y/x][z/x] : C} \qquad \textsc{MetaCont}\ \frac{\Gamma, \langle y\rangle : !A, \langle z\rangle : !A \vdash_d t : C}{\Gamma, \langle x\rangle : !A \vdash_d t[y/x][z/x] : C}$$

$$\rightarrow\textsc{Left}\ \frac{\Gamma, x : B \vdash_d t : C \qquad \Delta \vdash_d t' : A}{\Gamma, \Delta, f : A \to B \vdash_d t[x/f\ t'] : C} \qquad \rightarrow\textsc{Right}\ \frac{\Gamma, x : A \vdash_d t : B}{\Gamma \vdash_d \lambda x.t : A \to B}$$

$$\textsc{SoftPromotion}\ \frac{p_1 : A_1, \cdots, p_n : A_n \vdash_d t : B}{\langle p_1\rangle : !A_1, \cdots, \langle p_n\rangle : !A_n \vdash_d \langle t\rangle : !B} \qquad \textsc{Dereliction}\ \frac{\Gamma, p : A \vdash_d t : B}{\Gamma, \langle p\rangle : !A \vdash_d t : B}$$

$$\textsc{Digging}\ \frac{\Gamma, \langle\langle p\rangle\rangle : !!A \vdash_d t : B}{\Gamma, \langle p\rangle : !A \vdash_d t : B} \qquad \textsc{Cut}\ \frac{\Gamma \vdash_d t' : A \quad \Delta, x : A \vdash_d t : B}{\Gamma, \Delta \vdash_d t[x/t'] : B}$$

FIGURE 10.1 – règles de SIL_d

$$\textsc{SoftPromotion}\ \frac{\textsc{Digging}\ \dfrac{\langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n \vdash_d t : B}{\langle\langle x_1\rangle\rangle : !!A_1, \cdots, \langle\langle x_n\rangle\rangle : !!A_n \vdash_d \langle t\rangle : !B}}{\langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n \vdash_d \langle t\rangle : !B}$$

□

L'intérêt des motifs p est d'imbriquer les chevrons $\langle\rangle$. Un motifs $\langle\langle x\rangle\rangle$ peut alors passer deux fois la règles SoftPromotion, alors qu'auparavant, un variable ne pouvait en traverser qu'une.

La nouvelle règle Digging augmente le niveau d'imbrication des chevrons d'une variable de de Γ, en prévision du passage de plusieurs règles SoftPromotion. La règle Digging n'a semble t'il pas de contenu calculatoire. La règle Dereliction quant à elle décroit le nombre d'imbrications.

10.3 Conclusion

La restriction du système PIL au système SIL permet de mettre en évidence un usage de la déréliction dans le cadre des langages réflexifs. On y voit au moins deux intérêt : (i) la réflexion (la transformation d'une donnée en un programme) (ii) l'insertion d'une donnée dans une autre. Le système SIL est une restriction de l'usage de la déréliction au point (ii)

D'autre part, on a vu comment retrouver la puissance de PIL à partir de SIL. Comme remarqué par Lafont [Laf02], il suffit d'ajouter au système l'opérateur de digging.

Chapitre 11

Réification

Dans la Section 7.3.2, on a montré qu'il était impossible d'ajouter un opérateur de réification sans mettre à mal la confluence du système. Si on *fixe une stratégie de d'évaluation*, on peut à nouveau considérer le problème. De plus, on va *restraindre l'évaluation aux termes clos*. En effet, grâce à cette restriction on pourra à loisir transformer un terme t en terme $\langle t \rangle$: la règle PROMOTION s'appliquera en effet toujours.

En introduction, on va considérer le langage Λ_R étendu par un opérateur de réification *ad-hoc* . On montrera que dans le cadre d'une évaluation de terme clos, on peut ajouter le type $A \to !A$ à PIL_n.

La majorité du chapitre sera consacrée à l'extension de Λ_R en utilisant un opérateur de contrôle réifiant tout le contexte d'evaluation, à la façon de FELLEISEN.

Enfin, on montrera que l'opérateur de contrôle peux s'exprimer dans une machine de KRIVINE adaptée. On montrera que, là aussi, on peut typer les expressions dans le cas où les termes sont clos.

11.1 Introduction : réification *ad-hoc*

Syntaxe On définit une extension Λ_R^{box} de Λ_R

$$\Lambda_R \ni t \stackrel{\text{def}}{=} x \mid t\ t \mid \lambda x.t \mid \langle t \rangle \mid \mathsf{let}\ \langle x \rangle = t\ \mathsf{in}\ t \mid \mathsf{box}$$

On fixe une stratégie d'évaluation pour les deux raisons évoquées plus haut :

— On va parler de réification, donc on se place dans un langage avec stratégie.

— On veut limiter les réductions aux termes clos.

Liaison et substitutions On a juste ajouté une constante à Λ_R. La liaison des variables et les substitutions ne sont donc pas différentes de Λ_R.

Contextes Les *contextes d'évaluation* en stratégie CBN sont données par

$$\Phi = [] \mid \Phi\ t \mid \mathsf{let}\ \langle x \rangle = \Phi\ \mathsf{in}\ t$$

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash_{\mathsf{box}} x : A} \qquad \textsc{Abstraction}\ \frac{\Gamma, x : A \vdash_{\mathsf{box}} t : B}{\Gamma \vdash_{\mathsf{box}} \lambda x.t : A \to B}$$

$$\textsc{Application}\ \frac{\Gamma \vdash_{\mathsf{box}} t : A \to B \qquad \Gamma \vdash_{\mathsf{box}} t' : A}{\Gamma \vdash_{\mathsf{box}} t\ t' : B}$$

$$\textsc{Promotion}\ \frac{\langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n \vdash_{\mathsf{box}} t : C}{\langle x_1\rangle : !A_1, \cdots, \langle x_n\rangle : !A_n, y_1 : B_1, \cdots, y_m : B_m \vdash_{\mathsf{box}} \langle t\rangle : !C} \qquad \textsc{Dereliction}\ \frac{}{\Gamma, \langle x\rangle : !A \vdash_{\mathsf{box}} x : A}$$

$$\textsc{Let}\ \frac{\Gamma \vdash_{\mathsf{box}} t' : !A \qquad \Gamma, \langle x\rangle : !A \vdash_{\mathsf{box}} t : B}{\Gamma \vdash_{\mathsf{box}} \mathsf{let}\ \langle x\rangle = t'\ \mathsf{in}\ t : B} \qquad \textsc{Box}\ \frac{}{\Gamma \vdash_{\mathsf{box}} \mathsf{box} : A \to !A}$$

FIGURE 11.1 – Système de types de Λ_R^{box}

On remarquera les contextes usuels [] et $\Phi\ t$, d'usage normal pour une stratégie CBN. Par contre, la forme $\mathsf{let}\ \langle x\rangle = \Phi\ \mathsf{in}\ t$ est nouvelle. En effet, elle oblige la réduction de t dans le terme $\mathsf{let}\ \langle x\rangle = t\ \mathsf{in}\ t'$, ce qui est inabituel en stratégie CBN. Notre stratégie de réduction est un hybride entre CBN et CBV. La règle de réduction sera notée $\rightharpoonup$.

Les règles de réduction sont données par

$$\begin{array}{rcl} \Phi[(\lambda x.t)\ t'] & \overset{\lambda}{\rightharpoonup} & \Phi[t[x/t']] \\ \Phi[\mathsf{let}\ \langle x\rangle = \langle t'\rangle\ \mathsf{in}\ t] & \overset{\mathsf{let}}{\rightharpoonup} & \Phi[t[x/t']] \\ \Phi[\mathsf{box}\, t] & \overset{\mathsf{box}}{\rightharpoonup} & \Phi[\langle t\rangle] \end{array}$$

On pose $\rightharpoonup \overset{\text{def}}{=} \overset{\lambda}{\rightharpoonup} \cup \overset{\mathsf{let}}{\rightharpoonup} \cup \overset{\mathsf{box}}{\rightharpoonup}$ et $\rightharpoonup_{\text{head}}$ la restriction de $\rightharpoonup$ lorsque $\Phi = []$. La clôture transitive réflexive de $\rightharpoonup$ est notée $\overset{*}{\rightharpoonup}$. Par exemple, on a $(\lambda x.\, \mathsf{box}\, x)\ t \overset{*}{\rightharpoonup} \langle t\rangle$

Typage La langage peut-être typé en ajoutant à PIL_n le type $A \to !A$ de la constante box, c'est à dire qu'on ajoute la règle

$$\textsc{Box}\ \frac{}{\Gamma \vdash_{\mathsf{box}} \mathsf{box} : A \to !A}$$

Le système complet est donné dans la Figure 11.1. On peut montrer qu'il est toujours correct, c'est à dire qu'il continue de vérifier la propriété de Réduction du Sujet.

Lemme 11.1.1 (Substitution)**.** *Si* $\Gamma \vdash_{\mathsf{box}} t_1 : A$ *et* $\Gamma, x : A \vdash_{\mathsf{box}} t_2 : B$*, alors* $\Gamma \vdash_{\mathsf{box}} t_2[x/t_1] : B$*. Autrement dit, la règle*

$$\textsc{Cut}\ \frac{\Gamma \vdash_{\mathsf{box}} t_1 : A \qquad \Gamma, x : A \vdash_{\mathsf{box}} t_2 : B}{\Gamma \vdash_{\mathsf{box}} t_2[x/t_1] : B}$$

est admissible.

Démonstration. Par induction sur t_2. Le seul cas nouveau est lorsque $t_2 = \mathsf{box}$, et dans ce cas, $B = B' \to !B'$ et donc $\Gamma \vdash_{\mathsf{box}} \mathsf{box} : B' \to !B'$. □

Lemme 11.1.2 (Méta Substitution)**.** *Si* $\Gamma \vdash_{\mathsf{box}} \langle t_1 \rangle : !A$ *et* $\Gamma, \langle x \rangle : !A \vdash_{\mathsf{box}} t_2 : B$ *alors* $\Gamma \vdash_{\mathsf{box}} t_2[x/t_1] : B$*. Autrement dit, la règle*

$$\frac{\Gamma \vdash_{\mathsf{box}} \langle t_1 \rangle : !A \qquad \Gamma, \langle x \rangle : !A \vdash_{\mathsf{box}} t_2 : B}{\Gamma \vdash_{\mathsf{box}} t_2[x/t_1] : B}\ \textsc{MCut}$$

est admissible.

Démonstration. La preuve est la même que celle du Lemme 11.1.1 □

Lemme 11.1.3 (Compositionalité)**.** *Si* $\vdash_{\mathsf{box}} \Phi[t] : A$

- $\vdash_{\mathsf{box}} t : B$*,*
- *pour tout* t'*, si* $\vdash_{\mathsf{box}} t' : B$ *alors* $\vdash_{\mathsf{box}} \Phi[t'] : A$

Démonstration. Même démonstration que pour le Lemme 9.2.3. Par induction sur Φ.

- Si $\Phi = []$, alors $B \stackrel{\text{def}}{=} A$ fonctionne.
- Si $\Phi = \Phi'\ t_1$, alors par inversibilité, $\vdash_{\mathsf{box}} \Phi'[t] : C \to A$ et $\vdash_{\mathsf{box}} t_1 : C$. Par hypothèse d'induction, il existe B tel que $\vdash_{\mathsf{box}} t : B$. De plus, pour tout t' tel que $\vdash_{\mathsf{box}} t' : B$, $\vdash_{\mathsf{box}} \Phi'[t'] : C \to A$. Donc $\vdash_{\mathsf{box}} \Phi[t'] : A$ par la règle Application.
- Si $\Phi = \mathsf{let}\ \langle x \rangle = \Phi'\ \mathsf{in}\ t_1$, alors, par inversibilité, il existe C tel que $\langle x \rangle : !C \vdash_{\mathsf{box}} t_1 : A$ et $\vdash_{\mathsf{box}} \Phi'[t] : !C$. Par hypothèse d'induction, il existe B tels que $\vdash_{\mathsf{box}} t : B$. De plus, pour tout t' tel que $\vdash_{\mathsf{box}} t' : B$, $\vdash_{\mathsf{box}} \Phi'[t'] : !C$. Alors $\vdash_{\mathsf{box}} \Phi[t'] : A$ par la règle Let.

□

Lemme 11.1.4 (Réduction en tête)**.** *Si* $t \to_{\text{head}} t'$ *et* $\vdash_{\mathsf{box}} t : A$*, alors* $\vdash_{\mathsf{box}} t' : A$

Démonstration. Même démonstration que pour le Lemme 9.2.2. Par analyse des cas de t. Le seul cas nouveau est pour $t = \mathsf{box}\, t' \rightharpoonup \langle t' \rangle$. Dans ce cas, $\vdash_{\mathsf{box}} t : !A$ et $\vdash_{\mathsf{box}} t' : A$. Donc, par Promotion, $\vdash_{\mathsf{box}} \langle t' \rangle : !A$. *C'est ici qu'il est important de ne réduire que des termes clos.* □

Théorème 11.1.5 (Réduction du sujet)**.** *Si* $\vdash_{\mathsf{box}} t : A$ *et* $t \rightharpoonup t'$ *alors* $\vdash_{\mathsf{box}} t' : A$*.*

Démonstration. Même démonstration que pour le Théorème 8.2.5. Si $t \rightharpoonup t'$, il existe Φ, s, s' tel que $t = \Phi[s]$, $s \to_{\text{head}} s'$ et $t' = \Phi[s']$. Par le Lemme de Composition, $\vdash_{\mathsf{box}} s : B$. Par le Lemme 11.1.4, $\vdash_{\mathsf{box}} s' : B$. Par compositionnalité à nouveau, $\vdash_{\mathsf{box}} E[s'] = t' : A$. □

On a donné un premier exemple d'introduction de la réification dans Λ_R, dans cadre d'un langage avec stratégie d'évaluation. En cela l'opérateur box est semblable à l'opérateur *quote* de LISP, où l'utilisation de la stratégie CBN permet au langage de geler l'exécution d'un terme avant de le réduire.

D'autre part, le typage de box est possible car les termes que l'on réifie sont toujours clos, comme l'impose la stratégie CBN. Or tout terme clos peut être réifié par la règle Promotion.

11.2 Réification des contextes d'évaluation

11.2.1 Motivations

Une continuation est une façon explicite de spécifier où doit être retourné le résultat d'un calcul. Lorsqu'on réduit le terme $t = (\lambda x.x+x)2$ au terme 4, on suppose implicitement que le résultat de ce calcul sera utilisé par le contexte "autour" de t. Par exemple, on suppose que dans le terme $t + 5$, le terme t va retourner son résultat dans le contexte $[] + 5$.

Une continuation permet de s'affranchir de ce retour implicite dans le contexte courant, en spécifiant explicitement où sera retournée la valeur du résultat. Par exemple, si k_Φ est une continuation vers un contexte d'exécution Φ, le terme $(k_\Phi\ t) + 5$ ne va pas se réduire à $4 + 5$ mais en $\Phi[4]$. Le contexte $[] + 5$ est écrasé par Φ.

Dans sa thèse [Fel87], Felleisen décrit un opérateur $\mathcal{F}$ permettant de capturer le contexte d'exécution Φ où a lieu la réduction

$$\Phi[\mathcal{F}(t)] \to t\ (\lambda x.\Phi[x])$$

Des travaux sur la valeur logique d'un tel opérateur ont été faits par Griffin dans [Gri90] ou Parigot dans [Par92]. Ils notent la possibilité d'associer un calcul à la logique classique en utilisant des continuations.

D'autre part, Selinger, dans [Sel03] montre une implémentation du calcul de Parigot dans une machine de Krivine ([Kri07]). Les continuations apparaissent comme des objets de première classe.

Notre idée est d'utiliser cet accès aux continuations pour offrir à notre système une sorte de réification, jusqu'alors impossible. On ajoute un opérateur $\mathcal{C}$ renvoyant une version réifiée du contexte d'évaluation.

Notre approche ressemble à celle de [MF93]. Le langage décrit dans l'article a deux opérateurs, le premier (l'opérateur de réification) capturant une version réifiée du contexte d'évaluation courant, et un second (l'opérateur de réflexion) déclenchant l'exécution d'un contexte d'exécution réifié.

11.2.2 Définition du langage Λ_R^K

11.2.2.1 Syntaxe

L'ensemble de termes Λ_R^K est une extension de Λ_R selon la syntaxe suivante : on définit en même temps les termes t de Λ_R^K et les contextes d'évaluation Φ, les mêmes que dans la Section 11.1.

$$\begin{array}{rcl} \Lambda_R^K \ni t & \stackrel{\text{def}}{=} & x \mid \lambda x.t \mid t\ t \mid \langle t \rangle \mid \mathsf{let}\ \langle x \rangle = t\ \mathsf{in}\ t \mid k_\Phi \mid \mathcal{C} \\ \Phi & \stackrel{\text{def}}{=} & [] \mid \Phi\ t \mid \mathsf{let}\ \langle x \rangle = \Phi\ \mathsf{in}\ t \end{array}$$

Les deux nouveaux termes $\mathcal{C}$ et k_Φ sont deux termes (clos) permettant de manipuler les contextes d'évaluation. L'opérateur $\mathcal{C}$ renvoie une version réifiée du contexte courant. Le terme k_Φ désigne un contexte Φ et permet de continuer le calcul dans ce contexte.

$$\begin{array}{rcl}
\Phi[(\lambda x.t)\ t'] & \rightharpoonup & \Phi[t[x/t']] \\
\Phi[\mathsf{let}\ \langle x\rangle = \langle t'\rangle\ \mathsf{in}\ t] & \rightharpoonup & \Phi[t[x/t']] \\
\Phi[\mathcal{C}\, t] & \rightharpoonup & t\ \langle k_\Phi\rangle \\
\Phi[k_{\Phi'}\ t] & \rightharpoonup & \Phi'[t]
\end{array}$$

FIGURE 11.2 – Règles de réduction de Λ_R^K

Introduire un opérateur $\mathcal{C}$ capturant le contexte de réduction E rend le système non confluent. Par exemple, en utilisant l'opérateur de FELLEISEN, le terme $\mathcal{F}(\lambda k.(k\ 1)\ (k\ 2))$ peut se réduire à la fois à 1 et à 2.

11.2.2.2 Liaison des variables

Elle est tout à fait semblable à celle du langage Λ_R, sachant que $\mathcal{C}$, et k_Φ sont clos. Or, cette dernière condition impose également la fermeture des contextes Φ. Donc on doit définir $\mathrm{FV}^{\mathsf{let}}$ et FV^{λ} sur les contextes Φ.

$$\begin{array}{rcl}
\mathrm{FV}^{\lambda}([]) & \stackrel{\text{def}}{=} & \emptyset \\
\mathrm{FV}^{\lambda}(\Phi\ t) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\lambda}(\Phi) \cup \mathrm{FV}^{\lambda}(t) \\
\mathrm{FV}^{\lambda}(\mathsf{let}\ \langle x\rangle = \Phi\ \mathsf{in}\ t) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\lambda}(\Phi) \cup (\mathrm{FV}^{\lambda}(t) \setminus x) \\
\mathrm{FV}^{\lambda}(\mathcal{C}\ \ \Phi) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\lambda}(\Phi) \\
\mathrm{FV}^{\lambda}(k_{\Phi'}\ \Phi) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\lambda}(\Phi)
\end{array}$$

$$\begin{array}{rcl}
\mathrm{FV}^{\mathsf{let}}([]) & \stackrel{\text{def}}{=} & \emptyset \\
\mathrm{FV}^{\mathsf{let}}(\Phi\ t) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\mathsf{let}}(\Phi) \cup \mathrm{FV}^{\mathsf{let}}(t) \\
\mathrm{FV}^{\mathsf{let}}(\mathsf{let}\ \langle x\rangle = \Phi\ \mathsf{in}\ t) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\mathsf{let}}(\Phi) \cup (\mathrm{FV}^{\mathsf{let}}(t) \setminus x) \\
\mathrm{FV}^{\mathsf{let}}(\mathcal{C}\ \ \Phi) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\mathsf{let}}(\Phi) \\
\mathrm{FV}^{\mathsf{let}}(k_{\Phi'}\ \Phi) & \stackrel{\text{def}}{=} & \mathrm{FV}^{\mathsf{let}}(\Phi)
\end{array}$$

Dans la suite on considèrera que les contextes Φ sont tous clos, c'est à dire que $\mathrm{FV}^{\lambda}(\Phi) \cup \mathrm{FV}^{\mathsf{let}}(\Phi) = \emptyset$.

11.2.2.3 Règles de réduction

On définit la relation de transition $\rightharpoonup$ dans la Figure 11.2. Les termes k_Φ dénotent les *continuations*. Ce sont des constantes indexées par des contextes d'évaluation. Le terme $\mathcal{C}$ est l'*opérateur de contrôle*. Notre opérateur correspond à l'opérateur $\mathcal{F}$ dans le travail de FELLEISEN [Fel87 ; Gri90] (excepté le coté gèle de calculs).

La règle

$$\Phi[\mathcal{C}\, t] \rightharpoonup t\ \langle k_\Phi\rangle$$

donne le comportement de l'opérateur $\mathcal{C}$. Il capture l'environnement courant Φ et demande l'exécution du terme en paramètre t en lui passant une version réifiée de ce contexte $\langle k_\Phi\rangle$. C'est donc un opérateur de réification. D'autre part, la règle

$$\Phi[k_{\Phi'}\ t] \rightharpoonup \Phi'[t]$$

montre qu'on peut restaurer un contexte d'évaluation Φ' en appelant k'_Φ sur le prochain terme t à évaluer dans ce contexte.

Définition 11.2.1 (Valeurs)**.** Les *valeurs* v de Λ_R^K sont définies par la syntaxe suivante

$$v \stackrel{\text{def}}{=} \lambda x.t \mid \langle t \rangle \mid \mathcal{C} \mid k_\Phi$$

Propriété 11.2.2. *Les valeurs ne se réduisent pas par* $\rightharpoonup$.

Démonstration. Aucun contexte d'évaluation Φ n'est de la forme d'une valeur. □

11.2.2.4 Discussion sur les règles d'évaluation

L'opérateur de Felleisen vérifie

$$\Phi[\mathcal{F}\ t] \rightharpoonup t\ k_\Phi$$

alors que notre opérateur de contrôle $\mathcal{C}$ vérifie

$$\Phi[\mathcal{C}\ t] \rightharpoonup t\ \langle k_\Phi \rangle$$

L'opérateur $\mathcal{C}$ est donc différents de $\mathcal{F}$ car il retourne une version *réifiée* du contexte Φ. C'est pourquoi on a $t\ \langle k_\Phi \rangle$ à la place de $t\ k_\Phi$.

La règle de la continuation reste elle inchangée.

$$\Phi'[k_\Phi\ t] \rightharpoonup \Phi[t]$$

On peut retrouver l'opérateur l'expressivité de $\mathcal{F}$ à partir de $\mathcal{C}$. On pose pour cela $\mathcal{F} \stackrel{\text{def}}{=} \lambda t.\, \mathcal{C}(\lambda k.t\ (\mathsf{run}\, k))$. En effet

$$\begin{array}{rcl} \Phi[\mathcal{F}\ t] & \rightharpoonup & \Phi[\mathcal{C}(\lambda k.t\ (\mathsf{run}\, k))] \\ & \rightharpoonup & (\lambda k.t\ (\mathsf{run}\, k))\ \langle k_\Phi \rangle \\ & \rightharpoonup & t\ (\mathsf{run}\, \langle k_\Phi \rangle) \\ & \stackrel{*}{\rightharpoonup} & t\ k_\Phi \end{array}$$

11.2.2.5 Réification

Grâce à l'opérateur $\mathcal{C}$ on peut réifier tout terme t clos, à la manière du box de la Section 11.1. Posons

$$\mathsf{box}' \stackrel{\text{def}}{=} \lambda t.\, \mathcal{C}\ (\lambda k.(\mathcal{C}\ (\mathsf{run}\, k)\ t))$$

On a alors

$$\begin{array}{rcl} \Phi[\mathsf{box}'\, t] & \rightharpoonup & \Phi[\mathcal{C}\ (\lambda k.(\mathcal{C}\ (\mathsf{run}\, k)\ t))] \\ & \rightharpoonup & (\lambda k.(\mathcal{C}\ (\mathsf{run}\, k)\ t))\ \langle k_\Phi \rangle \\ & \rightharpoonup & \mathcal{C}\ (\mathsf{run}\, \langle k_\Phi \rangle)\ t \\ & \rightharpoonup & (\mathsf{run}\, \langle k_\Phi \rangle)\ \langle k_{[]t} \rangle \\ & \stackrel{*}{\rightharpoonup} & k_\Phi\ \langle k_{[]t} \rangle \\ & \stackrel{*}{\rightharpoonup} & \Phi[\langle k_{[]t} \rangle] \end{array}$$

Le terme $\mathsf{box}'\, t$ a donc *réifié* t en l'insérant dans un contexte.

$$
\begin{array}{llllllll}
t\ t' & \star & \Phi & \to & t & \star & \Phi \circ ([]\ t') & (\textsc{Push}) \\
\lambda x.t & \star & \Phi \circ ([]\ t') & \to & t[x/t'] & \star & \Phi & (\textsc{Pop}) \\
\mathsf{let}\ \langle x\rangle = t'\ \mathsf{in}\ t & \star & \Phi & \to & t' & \star & \Phi \circ (\mathsf{let}\ \langle x\rangle = []\ \mathsf{in}\ t) & (\textsc{LetWait}) \\
\langle t'\rangle & \star & \Phi \circ (\mathsf{let}\ \langle x\rangle = []\ \mathsf{in}\ t) & \to & t[x/t'] & \star & \Phi & (\textsc{LetRed}) \\
\mathcal{C} & \star & \Phi \circ ([]\ t) & \to & t & \star & []\ \langle k_\Phi\rangle & (\textsc{CRule}) \\
k_{\Phi'} & \star & \Phi \circ ([]\ t) & \to & t & \star & \Phi' & (\textsc{KRule})
\end{array}
$$

Figure 11.3 – Transition de la machine de Krivine

11.3 Expression dans une machine de Krivine

11.3.1 Sémantique de la machine

La stratégie CBN est bien connue ([Kri07 ; Sel03]) pour être interprétée par la Machine de Krivine. Un état de la machine est donné par le couple $t \star \Phi$, où t et Φ sont clos. On rappelle que l'on peut composer deux contextes d'évaluation Φ et Φ' en le contexte $\Phi \circ \Phi'$ en remplaçant le "trou" $[]$ de Φ par Φ' (Section 5.1.2).

Propriété 11.3.1. *Si* $\Phi_1 \circ \Phi = \Phi_2 \circ \Phi$ *alors* $\Phi_1 = \Phi_2$.

Démonstration. On sait déjà que Φ_1 et Φ_2 sont de même taille l. Par induction sur l,

— Si $l = 0$, alors $\Phi_1 = \Phi_2 = []$.

— Si $l = n+1$, alors Φ_1 et Φ_2 sont de la même forme $\Phi_1'\ t$ et $\Phi_2'\ t$ (resp. $\mathsf{let}\ \langle x\rangle = \Phi_1'\ \mathsf{in}\ t$ et $\mathsf{let}\ \langle x\rangle = \Phi_2'\ \mathsf{in}\ t$,). Donc $\Phi_1' \circ \Phi = \Phi_2' \circ \Phi$. Par induction $\Phi_1' = \Phi_2'$ et donc $\Phi_1 = \Phi_2$.

□

L'exécution de la machine (adaptée pour manipuler les expressions gelées) est donnée dans la Figure 11.3. Une réduction $t \star \Phi \to t' \star \Phi'$ simule une réduction $\Phi[t] \rightharpoonup \Phi'[t']$. Les règles Push et Pop sont les règles usuelles de la machine de Krivine ([Kri07]). On notera cependant qu'au lieu d'utiliser une pile π, on utilise un contexte d'évaluation Φ. Cette modification mineure est justifiée dans notre travail par notre volonté de cohérence avec les autres sections définissant des extensions du λ-calcul. Par exemple, l'environnement $\Phi \circ ([]\ t)$ correspond à la pile $t.\pi$, où π est la pile correspondant à Φ. Par la Propriété 11.3.1, on sait que la notation utilisant la composition $\circ$ est bien définie, c'est à dire qu'il n'existe pas deux contextes $\Phi_1 \neq \Phi_2$ tels que $\Phi_1 \circ \Phi = \Phi_2 \circ \Phi$.

Quatre nouvelles règles sont utilisées. La règle LetWait impose que t soit totalement réduit (si possible en un terme de la forme $\langle \cdots \rangle$). On force donc cette réduction en stockant dans le contexte l'information de qui a forcé cette réduction. La règle LetRed opère la réduction du redex $\mathsf{let}\ \langle x\rangle = \langle t'\rangle\ \mathsf{in}\ \langle t\rangle$.

La règle CRule capture le contexte courant, le réifie et le passe à son premier argument. La règle KRule rétablit un contexte.

L'*état initial* de la machine évaluant le terme clos t est $t \star []$. Les *états finaux* de la machines sont les états de la forme : $v \star []$. Ils correspondent à la fin d'une exécution normale. Tous les autres états d'arrêt sont des *états d'erreur*. Par exemple, $\lambda x.t \star \Phi \circ$

$(\mathsf{let}\,\langle x\rangle = [\,]\ \mathsf{in}\ t)$ est un état d'erreur. Les états d'erreur correspondent à des erreurs à l'exécution.

11.3.2 Exemple de calcul

On observe l'exécution du terme $\mathsf{box}'\ t$ dans la machine dans le contexte Φ

$$\begin{array}{rcl}
\mathsf{box}'\ t \star \Phi & \to & \mathsf{box}' \star \Phi \circ ([\,]\ t) \\
& \to & \mathcal{C}\ (\lambda k.(\mathcal{C}\ (\mathsf{run}\,k)\ t)) \star \Phi \\
& \to & \mathcal{C} \star \Phi \circ ([\,]\ (\lambda k.(\mathcal{C}\ (\mathsf{run}\,k)\ t))) \\
& \to & \lambda k.(\mathcal{C}\ (\mathsf{run}\,k)\ t) \star [\,]\ \langle k_\Phi\rangle \\
& \to & \mathcal{C}\ (\mathsf{run}\,\langle k_\Phi\rangle)\ t \star [\,] \\
& \to & \mathcal{C}\ (\mathsf{run}\,\langle k_\Phi\rangle) \star [\,]\ t \\
& \to & \mathcal{C} \star [\,]\ (\mathsf{run}\,\langle k_\Phi\rangle)\ t \\
& \to & (\mathsf{run}\,\langle k_\Phi\rangle) \star [\,]\ \langle k_{[]t}\rangle \\
& \to & \mathsf{run} \star [\,]\ \langle k_\Phi\rangle\ \langle k_{[]t}\rangle \\
& \to & \mathsf{let}\,\langle x\rangle = \langle k_\Phi\rangle\ \mathsf{in}\ x \star [\,]\langle k_{[]t}\rangle \\
& \to & \langle k_\Phi\rangle \star (\mathsf{let}\,\langle x\rangle = [\,]\ \mathsf{in}\ x)\langle k_{[]t}\rangle \\
& \to & k_\Phi \star \langle k_{[]t}\rangle \\
& \to & \langle k_{[]t}\rangle \star \Phi
\end{array}$$

On obtient le même résultat que dans la Section 11.2.2.5.

11.3.3 Correction du modèle de machine

Théorème 11.3.2. *Si la machine commence à l'état $t \star [\,]$*

1. *et s'arrête sur un état d'arrêt $t' \star \Phi'$ alors $t \xrightarrow{*} \Phi'[t']$ et $\Phi'[t']$ ne se réduit pas par $\rightharpoonup$.*
2. *et se réduit infiniment alors t se réduit infiniment par $\rightharpoonup$.*

La preuve utilise les lemmes suivants.

Lemme 11.3.3. *Si $t \star \Phi \to t' \star \Phi'$, alors $\Phi[t] = \Phi'[t']$ ou $\Phi[t] \rightharpoonup \Phi'[t']$.*

Démonstration. Par analyse de cas sur les règles de réduction de la machine. Les règles telles que $\Phi[t] = \Phi'[t']$ sont les règles Push et LetWait. Les règles Pop, LetRed, CRule et KRule vérifient $\Phi[t] \rightharpoonup \Phi'[t']$. □

Lemme 11.3.4. *Si $t \star \Phi$ ne se réduit pas, alors $\Phi[t]$ ne se réduit pas pour $\rightharpoonup$.*

Démonstration. Par analyse de cas sur les états d'arrêt qui sont

- $v \star [\,]$ ne se réduit pas et v non plus,
- $v \star \Phi \circ (\mathsf{let}\,\langle x\rangle = [\,]\ \mathsf{in}\ t')$ pour $v \neq \langle t\rangle$ ne se réduit pas et $\Phi[\mathsf{let}\,\langle x\rangle = v\ \mathsf{in}\ t']$ non plus,
- $\langle t\rangle \star \Phi \circ ([\,]\ t')$ ne se réduit pas et $\Phi[\langle t\rangle\ t']$ non plus.

□

Lemme 11.3.5. *Il n'existe pas de suite infinie $(t_i, \Phi_i)_{i\in\mathbb{N}}$ telle que pour tout i, $t_i \star \Phi_i \to t_{i+1} \star \Phi_{i+1}$ et $\Phi_i[t_i] = \Phi_{i+1}[t_{i+1}]$.*

Démonstration. Les seules règles telles que $t_i \star \Phi_i \to t_{i+1} \star \Phi_{i+1}$ et $\Phi_i[t_i] = \Phi_{i+1}[t_{i+1}]$ sont les règles Push et LetWait. Or ces deux règles augmentent strictement la taille du contexte : si $|\Phi|$ désigne la taille de Φ, alors $|\Phi_{i+1}| = |\Phi_i| + 1$. Or la suite $(\Phi_i[t_i])_i$ est constante et la constante est de taille finie. Donc la suite ne peux être infinie. □

On peut maintenant prouver le théorème

Démonstration. On démontre chaque point.

1. Si $t \star [] \xrightarrow{*} t' \star \Phi'$, alors $t \xrightarrow{*} \Phi'[t']$ (Lemme 11.3.3). De plus, si $t' \star \Phi'$ ne se réduit pas, alors $\Phi'[t']$ non plus (Lemme 11.3.4).
2. Si $t \star []$ se réduit infiniment, alors la réduction utilise une infinité de règles $t_1 \star \Phi_1 \to t_2 \star \Phi_2$ telles que $\Phi_1[t_1] \rightharpoonup \Phi_2[t_2]$ (Lemme 11.3.5). Donc t se réduit infiniment.

□

11.4 Système de types

11.4.1 Typage du λ-calcul avec continuations

Griffin dans [Gri90] montre comment l'opérateur de Felleisen ([Fel87]) permet de donner une interprétation de la Logique Classique dans le cadre de la correspondance de Curry-Howard. En effet, l'opérateur de Felleisen peut-être typé par la loi de la double négation

$$((A \to \bot) \to \bot) \to A$$

Or ajouter cette loi à la Logique Intuitionniste permet de retrouver l'expressivité de la Logique Classique. Dit autrement, une formule est prouvable en Logique Classique si et seulement si elle l'est en Logique Intuitionniste avec loi de la double négation [Joh37].

11.4.2 Réflexion et contrôle

On définit un nouveau système de types, PIL_n^k, inspiré par PIL_n. La syntaxe des types est le même que PIL mais on ajoute une constante $\bot$ aux constantes de type α.

$$A = \alpha \mid A \to A \mid !A$$

Le type $\bot$ désigne "le type vide", c'est à dire le type des termes qui ne retournent rien dans leur environnement immédiat. Par exemple, $k_\Phi\ \langle t \rangle$ doit être de type $\bot$ car il ne retourne rien localement, mais demande l'exécution de t dans le contexte Φ. On pourra noter $\sim A$ le type $A \Rightarrow \bot$.

Le système de type, PIL_n^k, est donné dans la Figure 11.4 et est adapté de PIL_n. Il utilise trois sortes de jugements (i) les jugements sur les termes, de la forme $\Gamma \vdash_k t : A$ (ii) les jugements sur les contextes, de la forme $\Phi : A \vdash_k$ (iii) les jugements sur les états, de la forme $\vdash_k t \star \Phi$. Les environnements de typage Γ sont de la même forme que dans les Chapitres 8 et 9, c'est à dire qu'ils sont composés d'hypothèses de la forme $x : A$ et de la forme $\langle x \rangle : !A$.

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash_k x : A} \qquad \textsc{Abstraction}\ \frac{\Gamma, x : A \vdash_k t : B}{\Gamma \vdash_k \lambda x.t : A \to B}$$

$$\textsc{Application}\ \frac{\Gamma \vdash_k t : A \to B \qquad \Gamma \vdash_k t' : A}{\Gamma \vdash_k t\ t' : B}$$

$$\textsc{Promotion}\ \frac{\langle x_1 \rangle : !A_1, \cdots, \langle x_n \rangle : !A_n \vdash_k t : C}{\langle x_1 \rangle : !A_1, \cdots, \langle x_n \rangle : !A_n, y_1 : B_1, \cdots, y_m : B_m \vdash_k \langle t \rangle : !C}$$

$$\textsc{Let}\ \frac{\Gamma \vdash_k t' : !A \qquad \Gamma, \langle x \rangle : !A \vdash_k t : B}{\Gamma \vdash_k \mathsf{let}\ \langle x \rangle = t'\ \mathsf{in}\ t : B} \qquad \textsc{Dereliction}\ \frac{}{\Gamma, \langle x \rangle : !A \vdash_k x : A}$$

$$\textsc{Call/cc}\ \frac{}{\Gamma \vdash_k \mathcal{C} : (!(A \to \bot) \to \bot) \to A} \qquad \textsc{Cont}\ \frac{\Phi : A \vdash_k}{\Gamma \vdash_k k_\Phi : A \to \bot}$$

$$\textsc{BotCTX}\ \frac{}{[] : \bot \vdash_k} \qquad \textsc{TopCTX}\ \frac{}{[] : A \vdash_k}$$

$$\textsc{AppCTX}\ \frac{\vdash_k t : A \qquad \Phi : B \vdash_k}{\Phi \circ ([]\ t) : A \to B \vdash_k} \qquad \textsc{LetCTX}\ \frac{\langle x \rangle : !A \vdash_k t : B \qquad \Phi : B \vdash_k}{\Phi \circ (\mathsf{let}\ \langle x \rangle = []\ \mathsf{in}\ t) : !A \vdash_k}$$

$$\textsc{Proc}\ \frac{\vdash_k t : A \qquad \Phi : A \vdash_k}{\vdash_k t \star \Phi}$$

FIGURE 11.4 – Système de types PIL_n^k

Comme dans l'opérateur $\mathcal{F}$ de Felleisen, notre opérateur $\mathcal{C}$ est typé par la règle de la double négation (règle Call/cc) à ceci près que notre négation utilise l'exponentielle : $\mathcal{C}$ est de type $(!(A \to \bot) \to \bot) \to A$.

Les règles sur les contextes sont les jugements de la forme $\Phi : A \vdash_k$. On notera que l'on n'a pas besoin de contexte de typage Γ car les contextes sont clos. On notera également la règle TopCTX où A est le type de tout le programme et BotCTX donnant le type $(\bot)$ de $[]$ dans le cas général (par exemple lorsque la machine se retrouve avec un contexte vide après l'utilisation de $\mathcal{C}$).

Finalement, on définit un jugement sur le bon typage d'un état de la machine : c'est la règle Proc. Pendant tout le déroulement du calcul, un état de la machine est un terme de type A faisant face à un contexte de type A (le type d'un contexte est le type de son "trou".)

Théorème 11.4.1. *Si t est clos et bien typé, alors $t \star []$ ne s'arrête pas sur un état d'erreur.*

Afin de prouver ce théorème, on utilise une version modifiée de la propriété de Réduction du Sujet qui affirme que la propriété de bon typage est conservée le long du calcul (Lemme 11.4.4). De plus, les états initiaux sont bien typés (Lemme 11.4.2). Et finalement, tous les états d'erreur sont mal typés (Lemme 11.4.3).

Lemme 11.4.2. *Si t est clos et s'il existe A tel que $\vdash_k t : A$, alors $\vdash_k t \star []$*

Démonstration. Trivial car $\vdash_k t : A$ et $[] : A \vdash_k$. □

Lemme 11.4.3. *Aucun état d'erreur n'est bien typé.*

Démonstration. Les états d'erreur sont

- $v \star \Phi \circ (\mathsf{let}\ \langle x \rangle = []\ \mathsf{in}\ t')$ pour $v \neq \langle t \rangle$ qui est mal typé car $\vdash_k v : \alpha$ ou $\vdash_k v : A \to B$ et $\Phi \circ (\mathsf{let}\ \langle x \rangle = []\ \mathsf{in}\ t') : !C \vdash_k$,
- $\langle t \rangle \star \Phi \circ ([]\ t')$ qui est mal typé car $\vdash_k \langle t \rangle : !A$ et $\Phi \circ ([]\ t') : B \to C \vdash_k$.

□

Le langage typé par PIL_n^k conserve la propriété de réduction du sujet.

Lemme 11.4.4 (Réduction du sujet)**.** *Si $t \star \Phi$ est bien typé et si $t \star \Phi \to t' \star \Phi$ alors $t' \star \Phi$ est bien typé.*

Démonstration. On a les Lemmes de Substitution 11.4.5 et 11.4.6. Le lemme est donc prouvé par analyse de cas sur la règle d'évaluation.

- Supposons que $\vdash_k t\ t' \star \Phi$. Alors $\vdash_k t : A \to B$, $\vdash_k t' : A$ et $\Phi : B$. Donc $\Phi \circ ([]\ t') : A \to B$. Donc $\vdash_k t \star \Phi \circ ([]\ t')$.
- Supposons que $\vdash_k \lambda x.t \star \Phi \circ ([]\ t')$. Alors $x : A \vdash_k t : B$, $\vdash_k t' : A$ et $\Phi : B \vdash_k$. Donc, par le Lemme de Substitution 11.4.5, $\vdash_k t[x/t'] : B$. Donc $\vdash_k t[x/t'] \star \Phi$.
- Supposons que $\mathsf{let}\ \langle x \rangle = t\ \mathsf{in}\ t' \star \Phi$ soit bien typé. Alors $\langle x \rangle : !A \vdash_k t' : B$ et $\vdash_k t : !A$ et $\Phi : B \vdash_k$. Alors $\Phi \circ (\mathsf{let}\ \langle x \rangle = []\ \mathsf{in}\ t') : !A \vdash_k$. Finalement $\vdash_k t \star \Phi \circ (\mathsf{let}\ \langle x \rangle = []\ \mathsf{in}\ t')$.

— Supposons que $\vdash_k \langle t\rangle \star \Phi \circ (\mathsf{let}\,\langle x\rangle = [] \;\mathsf{in}\; t')$. Alors $\vdash_k \langle t\rangle : !A$, $\langle x\rangle : !A \vdash_k t' : B$ et $\Phi : B \vdash_k$. Alors, par le Lemme de Substitution 11.4.6, $\vdash_k t'[x/t] : B$. Alors $\vdash_k t'[x/t] \star \Phi$.

— Supposons que $\vdash_k \mathcal{C} \star \Phi \circ ([]\ t)$. Alors $\vdash_k t : !(A \to \bot) \to \bot$ et $\Phi : A \vdash_k$. Donc $\vdash_k k_\Phi : A \to \bot$ et donc $[]\ \langle k_\Phi\rangle : !(A \to \bot) \to \bot \vdash_k$. Donc $\vdash_k t \star ([]\ \langle k_\Phi\rangle)$.

— Supposons que $\vdash_k k_{\Phi'} \star \Phi \circ ([]\ t)$. Alors $\Phi' : A \vdash_k$ et $\vdash_k t : A$. Donc $\vdash_k t \star \Phi'$.

□

On démontre à présent les lemmes de substitution

Lemme 11.4.5 (Substitution)**.** *Si* $\Gamma \vdash_k t_1 : A$ *et* $\Gamma, x : A \vdash_k t_2 : B$*, alors* $\Gamma \vdash_k t_2[x/t_1] : B$*. Autrement dit, la règle*

$$\frac{\Gamma \vdash_k t_1 : A \qquad \Gamma, x : A \vdash_k t_2 : B}{\Gamma \vdash_k t_2[x/t_1] : B}\ \textsc{Cut}$$

est admissible.

Démonstration. On a deux nouvelles constructions par rapport à PIL : $t_2 = \mathcal{C}$ et $t_2 = k_\Phi$. Ces deux termes sont clos (donc $t_2[x/t_1] = t_2$). Donc $\vdash_k t_2 : B$, et donc $\Gamma \vdash_k t_2[x/t_1] : B$. □

Lemme 11.4.6 (Méta Substitution)**.** *Si* $\Gamma \vdash_k \langle t_1\rangle : !A$ *et* $\Gamma, \langle x\rangle : !A \vdash_k t_2 : B$ *alors* $\Gamma \vdash_k t_2[x/t_1] : B$*. Autrement dit, la règle*

$$\frac{\Gamma \vdash_k \langle t_1\rangle : !A \qquad \Gamma, \langle x\rangle : !A \vdash_k t_2 : B}{\Gamma \vdash_k t_2[x/t_1] : B}\ \textsc{MCut}$$

est admissible.

Démonstration. La démonstration est la même que pour le Lemme 11.4.5. □

Exemple 11.4.7. Le terme $\mathsf{box}' = \lambda t.\mathcal{C}\ (\lambda k.(\mathcal{C}\ (\mathsf{run}\,k)\ t))$ est de type $A \to !(\neg\neg A)$. En effet

$$\dfrac{\dfrac{\dfrac{k : !(!((A \to \bot) \to \bot) \to \bot) \vdash_k \mathsf{run}\,k : !((A \to \bot) \to \bot) \to \bot}{k : !(!(\neg\neg A) \to \bot) \vdash_k \mathcal{C}\ (\mathsf{run}\,k) : A \to \bot} \qquad t : A \vdash_k t : A}{t : A, k : !(!(\neg\neg A) \to \bot) \vdash_k \mathcal{C}\ (\mathsf{run}\,k)\ t : \bot}}{\dfrac{t : A \vdash_k \lambda k.(\mathcal{C}\ (\mathsf{run}\,k)\ t) : !(!(\neg\neg A) \to \bot) \to \bot}{t : A \vdash_k \mathcal{C}\ (\lambda k.(\mathcal{C}\ (\mathsf{run}\,k)\ t)) : !(\neg\neg A)}}{\vdash_k \lambda t.\mathcal{C}\ (\lambda k.(\mathcal{C}\ (\mathsf{run}\,k)\ t)) : A \to !(\neg\neg A)}$$

(On laisse implicites les affaiblissements) ■

Exemple 11.4.8. L'opérateur $\mathcal{F} = \lambda t.\,\mathcal{C}(\lambda k.t\ (\mathsf{run}\,k))$ a bien le type de la double négation

$$\vdash_k \mathcal{F} : \neg\neg A \to A$$

En effet

$$\dfrac{\dfrac{\dfrac{\dfrac{t : \neg\neg A \vdash_k t : \neg A \to \bot \qquad k : !(A \to \bot) \vdash \mathsf{run}\,k : \neg A}{t : \neg\neg A, k : !(A \to \bot) \vdash_k t\ (\mathsf{run}\,k) :\to \bot}}{t : \neg\neg A \vdash_k \lambda k.t\ (\mathsf{run}\,k) : !(A \to \bot) \to \bot}}{t : \neg\neg A \vdash_k \mathcal{C}(\lambda k.t\ (\mathsf{run}\,k)) : A}}{\vdash_k \lambda t.\,\mathcal{C}(\lambda k.t\ (\mathsf{run}\,k)) : \neg\neg A \to A}$$

(On laisse implicites les affaiblissements) ■

11.4.3 Encodage de la Logique Classique

Il était possible d'encoder la Logique Intuitionniste dans le système de types PIL (Section 9.5). Maintenant on va montrer de la même manière qu'il est possible d'encoder la Logique Classique dans Λ_R^K en utilisant la même traduction (celle de Girard [Gir87]).

On rappelle dans un premier temps que l'ajout de la règle de double négation $((A \to \bot) \to \bot) \to A$ (où $\bot$ est un type atomique) à la Logique Intuitionniste permet de retrouver l'expressivité de la Logique Classique ([Joh37]). Par "ajouter", on veut dire ajouter la règle

$$\dfrac{}{\Gamma \vdash_n \neg\neg A \to A}$$

au système de la Logique Intuitionniste en déduction naturelle [7] en ayant posé $\neg A \stackrel{\text{def}}{=} A \to \bot$. L'opérateur de Felleisen $\mathcal{F}$ vérifie ([Gri90])

$$\dfrac{}{\Gamma \vdash_n \mathcal{F} : \neg\neg A \to A}$$

On pose $\mathcal{F}^g \stackrel{\text{def}}{=} \lambda t.\,\mathcal{C}(\lambda K.\,\mathsf{let}\,\langle k \rangle = K\ \mathsf{in}\ (\mathsf{run}\,t)\ \langle \lambda x.k\ (\mathsf{run}\,x)\rangle)$.

Propriété 11.4.9. *Le jugement* $\vdash_k \mathcal{F}^g : \sim\sim A \Rightarrow A$ *est dérivable.*

Démonstration. Voici un dérivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{t : !\sim\sim A \vdash_k \mathsf{run}\,t : !\sim A \to \bot \qquad \dfrac{\dfrac{\dfrac{\langle k \rangle : !(A \to \bot) \vdash_k k : A \to \bot \qquad x : !A \vdash_k \mathsf{run}\,x : A}{\langle k \rangle : !(A \to \bot), x : !A \vdash_k k\ (\mathsf{run}\,x) : \bot}}{\langle k \rangle : !(A \to \bot) \vdash_k \lambda x.k\ (\mathsf{run}\,x) : \sim A}}{\langle k \rangle : !(A \to \bot) \vdash_k \langle \lambda x.k\ (\mathsf{run}\,x)\rangle : !\sim A}}{t : !\sim\sim A, \langle k \rangle : !(A \to \bot) \vdash_k (\mathsf{run}\,t)\ \langle \lambda x.k\ (\mathsf{run}\,x)\rangle : \bot}}{t : !\sim\sim A, K : !(A \to \bot) \vdash_k \mathsf{let}\,\langle k \rangle = K\ \mathsf{in}\ (\mathsf{run}\,t)\ \langle \lambda x.k\ (\mathsf{run}\,x)\rangle : \bot}}{t : !\sim\sim A \vdash_k \lambda K.\,\mathsf{let}\,\langle k \rangle = K\ \mathsf{in}\ (\mathsf{run}\,t)\ \langle \lambda x.k\ (\mathsf{run}\,x)\rangle : !(A \to \bot) \to \bot}}{t : !\sim\sim A \vdash_k \mathcal{C}(\lambda K.\,\mathsf{let}\,\langle k \rangle = K\ \mathsf{in}\ (\mathsf{run}\,t)\ \langle \lambda x.k\ (\mathsf{run}\,x)\rangle) : A}}{\vdash_k \lambda t.\,\mathcal{C}(\lambda K.\,\mathsf{let}\,\langle k \rangle = K\ \mathsf{in}\ (\mathsf{run}\,t)\ \langle \lambda x.k\ (\mathsf{run}\,x)\rangle) : \sim\sim A \Rightarrow A}$$

7. Il est aussi possible de le faire en calcul des séquents en ajoutant l'axiome $\vdash_s \neg\neg A \to A$

□

Théorème 11.4.10. *Le jugement* $x_1 : A_1, \cdots, x_n : A_n \vdash_n t : A$ *est valide en Logique Intuitionniste avec double négation si et seulement si* $\langle x_1 \rangle : !A_1^g, \cdots, \langle x_n \rangle : !A_n^g \vdash_k t^g : A^g$ *est valide.*

Démonstration. Il suffit de compléter la preuve du Théorème 9.5.1 par le cas de la double négation. Comme les deux séquents $\Gamma \vdash_n \mathcal{F} : \neg\neg A \to A$ et $\langle \Gamma^g \rangle \vdash_n^{\text{PIL}} \mathcal{F}^g : \sim\sim A \Rightarrow A$ sont tous les deux valides (Propriété 11.4.9 pour le deuxième jugement), l'équivalence tient toujours. □

11.4.4 Cas de SIL

On peut se demander ce que vaut le système PIL_n^k en remplaçant la règle de promotion par la règle de soft-promotion. Le Théorème 11.4.1 et La propriété de réduction du sujet sont encore vraie. Finalement, le Théorème 11.4.10 n'est plus valable, pour la même raison que le Théorème 9.5.1 ne l'était pas dans SIL.

11.5 Conclusion

On a donné une extension du langage Λ_R permettant la réification du contexte d'exécution du programme. L'opération s'inspire du travail de Felleisen sur le lambda calcul avec captures de continuations.

De plus, on a donné une machine s'inspirant des machines de Krivine exprimant les réductions de notre langage. Cette machine est correcte et complète.

Notre langage est à nouveau typé. On montre que cette extension permet d'encoder la Logique Classique dans notre système de types.

On a étudié plus en détail le problème de la réification. La solution abordée ici consiste à choisir une stratégie d'exécution n'évaluant que les termes clos (par exemple CBN ou CBV). Cette contrainte permet de réifier n'importe quelle terme en utilisant la règle Promotion. Les deux solutions présentées utilisent cette simplification.

Conclusion

Contributions

Formalisation de phénomènes réflexifs La première contribution de cette thèse a été la formalisation de certains concepts des langages réflexifs (Chapitre 2). Certains ont été évoqués dans le travail de Smith [Smi83] (réflexion, réification, réflexion structurelle et procédurale), d'autres (vagues), plus récemment dans le travail de Reynaud [Rey10]. On a pour cela utilisé le formalisme des langages de programmation à la Jones.

L'idée d'un langage réflexif repose sur deux notions importantes : la notion d'interpréteur et le Théorème de Récursion de Kleene. Un interpréteur permet d'exécuter un programme vu comme une donnée, tandis que le Théorème de Récursion fourni des programmes ayant accès à leur propre code. On a combiné ces deux notions en une en définissant les *interpréteurs récursifs*, exécutant une donnée tout en fournissant le code de l'interpréteur récursif à la donnée.

On a implémenté ces différents concepts dans une machine abstraite (Chapitre 3). La particularité de cette machine est qu'elle est centrée sur la dualité programmes/données. Les opérations réflexives correspondent à des mouvements "physiques" de portions de codes de la zone des programmes à la zone des données. Cette "réflexion structurelle" nous a permis d'établir un exécution abstraite (inspirée de [Rey10]) segmentant le calcul en fonction des réflexions (Chapitre 4).

Réflexion dans un langage fonctionnel On a cherché à exprimer la réflexion dans un langage fonctionnel (λ-calcul) (Chapitre 7). La distinction entre programmes et données se fait au niveau de la capacité d'un terme à se réduire. On dira qu'un terme est une donnée s'il ne se réduit pas. L'idée est donc d'ajouter dans la syntaxe du λ-calcul un moyen de dire que la réduction d'un terme est *gelée*. À partir de cela, tous les termes gelés sont considérés comme des données, les autres comme des programmes. Cette idée n'est pas neuve est a déjà été utilisée dans les travaux sur la *staged computation* [Dav96 ; DP96]. Elle est historiquement basée sur les logiques temporelles et modales.

Le gel de l'exécution des termes pose néanmoins des difficultés, notamment pour garantir la confluence du langage. Des précautions sont à prendre lors de liaisons des variables d'un terme gelé. On obtient alors un langage confluent, comme espéré dans un langage fonctionnel. Cependant, la propriété de confluence interdit le principe de réification (transformation d'un programme en une donnée, ou encore, dans le cas de notre langage, d'un terme à un terme gelé).

On a donc étudié le problème de la réification en mettant de coté la propriété de

confluence (Chapitre 11). On a choisi une stratégie de réduction CBN. Il est alors possible de parler de réification (c'est ce que font des langages comme LISP où il existe une fonction *quote* stoppant la réduction d'un terme). On s'est alors intéressé la réification de toute le contexte d'évaluation, en s'inspirant des travaux de Felleisen [Fel87] sur le contrôle.

Interprétation logique Au niveau logique, un langage réflexif peut être exprimé dans le cadre de la logique linéaire. On utilise l'exponentielle ! comme type des termes gelé. Deux systèmes sont présentés : l'un utilisant la toute les capacités de la Logique Linéaire (Chapitre 9) et l'autre un sous-système strict, la Soft Linear Logic (Chapitre 8). On montre en quoi SIL est strictement inclus dans PIL, et comment retrouver les capacités de PIL à partir de SIL.

La distinction des deux systèmes permet d'interpréter le rôle des règles de la Logique Linéaire dans le cadre des programmes réflexifs (Chapitre 10). D'autre part, on peut comparer des problèmes typiquement sémantiques comme la cohérence d'une syntaxe de la Logique Linéaire [Wad92 ; Wad93] aux problèmes de confluence de notre langage. Enfin, notre système peux encoder les calculs intuitionnistes (λ-calcul simplement typé) et classique (avec l'opérateur de Felleisen typé par Griffin [Gri90]) en utilisant la traduction de Girard qui met en évidence la distinction programmes/données dans un terme (les paramètres sont des données (termes gelés), le reste est un programme).

On a montré quelques liens entre deux mondes apparemment éloignés, la Logique Linéaire et les langages réflexifs. Nous pensons que ces liens peuvent encore être approfondis.

Perspectives

Il reste des choses à faire à partir de Λ_R. Notamment, au niveau de l'expression dans le langage d'une sémantique intentionnelle. Dans le modèle actuel, seule la sémantique extensionnelle (comment se comporte le programme) des données est étudiée, c'est à dire que les résultats obtenus placent la donnée en position d'exécution. On vérifie par exemple que lors de son exécution, elle ne produira pas d'erreur, restera du même type, etc. Au contraire, la sémantique intentionnelle (comment a été écrit le programme), c'est à dire l'utilisation de la donnée en tant qu'objet syntaxique, n'a pas du tout été abordée. On aurait par exemple souhaité avoir un langage assez expressif pour écrire des programmes capables de changer leurs comportements en fonction de la forme syntaxique des données passées en paramètre.

Au niveau logique, on a montré que le λ-calcul simplement typé suffisait à typer les termes de Λ_R si on lui ajoutait l'exponentiel !. Cette modalité est une comonade. Nous pensons qu'il peut être intéressant de replacer cette comonade dans le contexte de la théorie des catégories, et d'essayer d'interpréter les résultats de cette théorie dans le domaine des langages réflexifs. L'une des idées est d'utiliser la décomposition de la modalité ! en deux foncteurs adjoints [Mel06]. Par exemple, la négation $\neg$ est un foncteur auto-adjoint : que se passerait il si on remplaçait $!A$ par $\neg\neg A$? Aurait-on alors une interprétation de $\langle t \rangle$ de type $\neg\neg A$ comme une continuation attendant une continuation ? Pour répondre à ces questions, on pourrait s'inspirer du travail de Melliès et Tabareau [MT10].

Bibliographie

[Ben+92] Nick Benton et al. *Term Assignment for Intuitionistic Linear Logic*. Rapp. tech. University of Cambridge, 1992.

[Ben95] P. N. Benton. "A mixed linear and non-linear logic : Proofs, terms and models". In : *Computer Science Logic : 8th Workshop, CSL '94 Kazimierz, Poland, September 25–30, 1994 Selected Papers*. Sous la dir. de Leszek Pacholski et Jerzy Tiuryn. Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, p. 121–135. isbn : 978-3-540-49404-1. doi : 10.1007/BFb0022251. url : http://dx.doi.org/10.1007/BFb0022251.

[Bie94] G M Bierman. "On intuitionistic linear logic". Thèse de doct. University of Cambridge, 1994.

[Bie95] G. M. Bierman. "What is a categorical model of Intuitionistic Linear Logic ?" In : *Typed Lambda Calculi and Applications : Second International Conference on Typed Lambda Calculi and Applications, TLCA '95 Edinburgh, United Kingdom, April 10–12, 1995 Proceedings*. Sous la dir. de Mariangiola Dezani-Ciancaglini et Gordon Plotkin. Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, p. 78–93. isbn : 978-3-540-49178-1. doi : 10.1007/BFb0014046. url : http://dx.doi.org/10.1007/BFb0014046.

[BM04] Patrick Baillot et Virgile Mogbil. "Soft lambda-calculus : a language for polynomial time computation". In : *In Proc. FoSSaCS, Springer LNCS 2987*. Springer, 2004, p. 27–41.

[Bon+12] Guillaume Bonfante et al. "Code synchronization by morphological analysis". In : *Malware* (2012).

[Bon+13] Guillaume Bonfante et al. "Analysis and Diversion of Duqu's Driver". In : *Malware* (2013).

[BP01] G M Bierman et V C V de Paiva. "On an Intuitionistic Modal Logic". In : *Studia Logica* 65 (2001), p. 2000.

[BT+91] Val Breazu-Tannen et al. "Inheritance As Implicit Coercion". In : *Inf. Comput.* 93.1 (juil. 1991), p. 172–221. issn : 0890-5401. doi : 10.1016/0890-5401(91)90055-7. url : http://dx.doi.org/10.1016/0890-5401(91)90055-7.

[CR73] Stephen Arthur Cook et Robert Allen Reckhow. "Time bounded random access machines". In : *Journal of Computer and System Sciences* 7.4 (1973), p. 354–375.

[CTL98] Christian Collberg, Clark Thomborson et Douglas Low. “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”. In : *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’98. San Diego, California, USA : ACM, 1998, p. 184–196. isbn : 0-89791-979-3. doi : 10.1145/268946.268962. url : http://doi.acm.org/10.1145/268946.268962.

[Dav96] Rowan Davies. “A temporal-logic approach to binding-time analysis”. In : *Logic in Computer Science* (1996), p. 184–195. issn : 1043-6871. doi : 10.1109/LICS.1996.561317.

[DMM88] Olivier Danvy, Karoline Malmkj&aelig ;r et Karoline Malmkjr. “Intensions and Extensions in a Reflective Tower”. In : *In Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. ACM Press, 1988, p. 327–341.

[DP96] Rowan Davies et Frank Pfenning. “A Modal Analysis of Staged Computation”. In : *Principles of programming languages* (1996).

[ER64] Calvin C. Elgot et Abraham Robinson. “Random-access Stored-Program Machines, an Approach to Programming Languages”. In : *Journal of the Association for Computing Machinery* 11.4 (1964), p. 365–399.

[Fel87] Matthias Felleisen. “The Calculi of Lambda-v-CS Conversion : a Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages”. Thèse de doct. Indiana University, 1987.

[Fut83] Yoshihiko Futamura. “Partial computation of programs”. In : *RIMS Symposia on Software Science and Engineering : Kyoto, 1982 Proceedings*. Sous la dir. d’Eiichi Goto et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 1983, p. 1–35. isbn : 978-3-540-39442-6. doi : 10.1007/3-540-11980-9_13. url : http://dx.doi.org/10.1007/3-540-11980-9_13.

[Gar70] M. Gardner. “The fantastic combinations of John Conway’s new solitaire game “life””. In : *Scientific American* 223 (oct. 1970), p. 120–123.

[Gen35] G. Gentzen. “Untersuchungen über das logische Schließen I”. In : *Mathematische Zeitschrift* 39 (1935), p. 176–210. url : http://eudml.org/doc/168546.

[Gir87] Jean-Yves Girard. “Linear logic”. In : *Theoretical Computer Science* 50.1 (1987), p. 1 –101. issn : 0304-3975. doi : http://dx.doi.org/10.1016/0304-3975(87)90045-4. url : http://www.sciencedirect.com/science/article/pii/0304397587900454.

[GJ91] Carsten K. Gomard et Neil D. Jones. “A partial evaluator for the untyped lambda-calculus”. In : *Journal of Functional Programming* 1.1 (1991), p. 21–69. doi : 10.1017.

[GJM12] Roberto Giacobazzi, Neil D. Jones et Isabella Mastroeni. "Obfuscation by Partial Evaluation of Distorted Interpreters". In : *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. PEPM '12. Philadelphia, Pennsylvania, USA : ACM, 2012, p. 63–72. ISBN : 978-1-4503-1118-2. DOI : 10.1145/2103746.2103761. URL : http://doi.acm.org/10.1145/2103746.2103761.

[Gri90] Timothy G. Griffin. "A Formulae-as-type Notion of Control". In : *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA : ACM, 1990, p. 47–58. ISBN : 0-89791-343-4. DOI : 10.1145/96709.96714. URL : http://doi.acm.org/10.1145/96709.96714.

[GRR07] Marco Gaboardi, Simona Ronchi et Della Rocca. *A soft type assignment system for λ-calculus*. 2007.

[GTL89] Jean-Yves Girard, Paul Taylor et Yves Lafont. *Proofs and Types*. New York, NY, USA : Cambridge University Press, 1989. ISBN : 0-521-37181-3.

[Har71] Juris Hartmanis. "Computational Complexity of Random Access Stored Program Machines". In : *Mathematical Systems Theory* (1971).

[Hin64] J. Roger Hindley. "The Church-Rosser Property and a Result in Combinatory Logic". Thèse de doct. University of Newcastle, 1964.

[How80] William Alvin Howard. "The formulas-as-types notion of construction". In : *Essays on Combinatory Logic, Lambda Calculus, and Formalism* (1980).

[JGS93] Neil D. Jones, Carsten K. Gomard et Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1993. ISBN : 0-13-020249-5.

[Joh37] Ingebrigt Johansson. "Der Minimalkalkül, ein reduzierter intuitionistischer Formalismus". In : *Compositio Mathematica* 4 (1937), p. 119–136.

[Jon97] Neil D. Jones. *Computability and Complexity*. MIT Press, 1997.

[JS86] Ulrik Jørring et William L. Scherlis. "Compilers and Staging Transformations". In : *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida : ACM, 1986, p. 86–96. DOI : 10.1145/512644.512652. URL : http://doi.acm.org/10.1145/512644.512652.

[Kah87] Gilles Kahn. "Natural semantics". In : *STACS 87 : 4th Annual Symposium on Theoretical Aspects of Computer Science Passau, Federal Republic of Germany, February 19–21, 1987 Proceedings*. Sous la dir. de Franz J. Brandenburg, Guy Vidal-Naquet et Martin Wirsing. Berlin, Heidelberg : Springer Berlin Heidelberg, 1987, p. 22–39. ISBN : 978-3-540-47419-7. DOI : 10.1007/BFb0039592. URL : http://dx.doi.org/10.1007/BFb0039592.

[Kle38] S. C. Kleene. "On Notation for Ordinal Numbers". In : *J. Symbolic Logic* 3.4 (déc. 1938), p. 150–155. URL : http://projecteuclid.org/euclid.jsl/1183385485.

[Kri07] Jean-Louis KRIVINE. “A Call-by-name Lambda-calculus Machine”. In : *Higher Order Symbol. Comput.* 20.3 (sept. 2007), p. 199–207. ISSN : 1388-3690. DOI : 10.1007/s10990-007-9018-9. URL : http://dx.doi.org/10.1007/s10990-007-9018-9.

[Laf02] Yves LAFONT. “Soft Linear Logic and Polynomial Time”. In : *Theoretical Computer Science* 318 (2002), p. 2004.

[Laf88] Yves LAFONT. “Logiques, catégories et machines”. Thèse de doct. Université Paris 7, 1988.

[LS91] Yves LAFONT et Thomas STREICHER. “Games semantics for linear logic”. In : *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. 1991, p. 43–50. DOI : 10.1109/LICS.1991.151629.

[Mar12] Jean-Yves MARION. “From Turing machines to computer viruses”. In : *Philosophical Transactions of the Royal Society of London A : Mathematical, Physical and Engineering Sciences* 370.1971 (2012), p. 3319–3339. ISSN : 1364-503X. DOI : 10.1098/rsta.2011.0332.

[Mel03] Paul-André MELLIÈS. “Categorical Models of Linear Logic Revisited”. In : 2003.

[Mel06] Paul-André MELLIÈS. “Functorial boxes in string diagrams”. In : *Computer Science Logic* (2006).

[MF93] Anurag MENDHEKAR et Dan FRIEDMAN. “Towards a theory of reflexive programming languages”. In : *Worskshop on Reflection and Meta-level Architectures* (1993).

[MM94] Simone MARTINI et Andrea MASINI. “A Modal View of Linear Logic”. English. In : *The Journal of Symbolic Logic* 59.3 (1994), pp. 888–899. ISSN : 00224812. URL : http://www.jstor.org/stable/2275915.

[Mos06] Lawrence S. MOSS. “Recursion Theorems and Self-Replication Via Text Register Machine Programs”. In : *European Association for Theoretical Computer Science* (2006).

[Mos10] Yiannis N. MOSCHOVAKIS. “Kleene’s amazing second recursion theorem”. In : *The Bulletin of Symbolic Logic* 16.2 (2010), p. 189–239. ISSN : 10798986. URL : http://www.jstor.org/stable/27805176.

[MT10] Paul-André MELLIÈS et Nicolas TABAREAU. “Resource modalities in tensor logic”. In : *Annals of Pure and Applied Logic* 161.5 (2010). The Third workshop on Games for Logic and Programming Languages (GaLoP)Galop 2008, p. 632–653. ISSN : 0168-0072. DOI : 10.1016/j.apal.2009.07.018. URL : http://www.sciencedirect.com/science/article/pii/S0168007209001602.

[Neu66] John von NEUMANN. *Theory of Self-Reproducing Automata*. Sous la dir. d’Arthur W. BURKS. Champaign, IL, USA : University of Illinois Press, 1966.

[NN92] Flemming NIELSON et Hanne Riis NIELSON. *Two-level Functional Languages*. New York, NY, USA : Cambridge University Press, 1992. ISBN : 0-521-40384-7.

[Odi89] Piergiorgio Odifreddi. *Classical Recursion Theory : The Theory of Functions and Sets of Natural Numbers*. Sole Distributors for the Usa et Canada, Elsevier Science Pub. Co., 1989.

[Par92] Michel Parigot. "$\lambda\mu$-calculus : an algorithmic interpretation of classical natural deduction". In : *Logic Programming and Automated Reasoning* (1992).

[Plo81] G. D. Plotkin. *A Structural Approach to Operational Semantics*. 1981.

[Rey10] Daniel Reynaud. "Analyse de codes auto-modifiants pour la sécurité informatique". Thèse de doct. INPL, 2010.

[Rog87] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability*. Cambridge, MA, USA : MIT Press, 1987. isbn : 0-262-68052-1.

[Ros73] Barry K. Rosen. "Tree-Manipulating Systems and Church-Rosser Theorems". In : *J. ACM* 20.1 (jan. 1973), p. 160–187. issn : 0004-5411. doi : 10.1145/321738.321750. url : http://doi.acm.org/10.1145/321738.321750.

[RR97] Simona Ronchi della Rocca et Luca Roversi. "Lambda Calculus and Intuitionistic Linear Logic". In : *Studia Logica* 59.3 (1997), p. 417–448. issn : 1572-8730. doi : 10.1023/A:1005092630115. url : http://dx.doi.org/10.1023/A:1005092630115.

[Sch96] Harold Schellinx. "A Linear Approach to Modal Proof Theory". English. In : *Proof Theory of Modal Logic*. Sous la dir. d'Heinrich Wansing. T. 2. Applied Logic Series. Springer Netherlands, 1996, p. 33–43. isbn : 978-90-481-4720-5. doi : 10.1007/978-94-017-2798-3_3. url : http://dx.doi.org/10.1007/978-94-017-2798-3_3.

[See89] R.A.G. Seely. "Linear Logic, $\star$-Autonomous Categories and Cofree Coalgebras". In : *In Categories in Computer Science and Logic*. American Mathematical Society, 1989, p. 371–382.

[Sel03] Peter Selinger. "From Continuation Passing Style to Krivine Abstract Machine". 2003.

[Smi83] Brian Cantwell Smith. "Reflection and Semantics in LISP". In : *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA : ACM, 1983, p. 23–35. isbn : 0-89791-125-3. doi : 10.1145/800017.800513. url : http://doi.acm.org/10.1145/800017.800513.

[Smu84] Raymond M. Smullyan. "Chameleonic Languages". In : *D. Reidel Publishing Compagny* (1984).

[Thi15] Aurélien Thierry. "Désassemblage et détection de logiciels malveillants automodifiants". Thèse de doct. Université de Lorraine, 2015.

[TS97] Walid Taha et Tim Sheard. "Multi-stage Programming with Explicit Annotations". In : *Partial Evaluation and Semantics-Based Program Manipulation* 32.12 (déc. 1997), p. 203–217. issn : 0362-1340. doi : 10.1145/258994.259019. url : http://doi.acm.org/10.1145/258994.259019.

[Wad92] Philip WADLER. "There's no substitute for linear logic". In : *International Workshop on the Mathematical Foundations of Programming Semantics* (1992).

[Wad93] Philip WADLER. "A syntax for linear logic". In : *International Conference on the Mathematical Foundations of Programming Semantics* (1993).

[WF] Mitchell WAND et Daniel P. FRIEDMAN. "The mystery of the tower revealed : A nonreflective description of the reflective tower". In : *LISP and Symbolic Computation* 1.1 (), p. 11–38. ISSN : 1573-0557. DOI : 10.1007/BF01806174. URL : http://dx.doi.org/10.1007/BF01806174.

[WF94] A.K. WRIGHT et M. FELLEISEN. "A Syntactic Approach to Type Soundness". In : *Inf. Comput.* 115.1 (nov. 1994), p. 38–94. ISSN : 0890-5401. DOI : 10.1006/inco.1994.1093. URL : http://dx.doi.org/10.1006/inco.1994.1093.

Résumé

Le but de cette thèse est de trouver des modèles de haut niveau dans lesquelles l'auto-modification s'exprime facilement.

Une donné est lisible et modifiable, alors qu'un programme est exécutable. On décrit une machine abstraite où cette dualité est structurellement mise en valeur. D'une part une zone de programmes contient tous les registres exécutables, et d'autre part une zone de données contient les registres lisibles et exécutables. L'auto-modification est permise par le passage d'un registre d'une zone à l'autre. Dans ce cadre, on donne une abstraction de l'exécution de la machine qui extrait seulement les informations d'auto-modification.

Logiquement, on essaye de trouver une correspondance de Curry-Howard entre un langage avec auto-modification et un système logique. Dans ce but on construit une extension de lambda-calcul avec termes gelés, c'est à dire des termes qui ne peuvent se réduire. Ces termes sont alors considérés comme des données, et les autres sont les programmes. Notre langage a les propriétés usuelles du lambda-calcul (confluence). D'autre part, on donne un système de types dans lequel un sous ensemble des termes du langage peuvent s'exprimer. Ce système est inspiré de la Logique Linéaire, sans gestion des ressources. On prouve que ce système de types a de bonnes propriétés, comme celle de la réduction du sujet. Finalement, on étant le système avec les continuations et la double négation, dans un style à la Krivine.

Mots clefs : virologie, auto-modification, obfuscation, sémantique des langages, réflexion, logique linéaire

Abstract

The goal of my Ph.D. is to finds high level models in which self-modification can be expressed.

What is readable and changeable is a *data*, and a *program* is executable. We propose an abstract machine where this duality is *structurally* emphasized. On one hand the program zone beholds registers which can be executed, and on the other hand data zone contains readable and changeable registers. Self-modification is enabled by passing a data register into program zone, or a program register into data zone. In this case, we give an abstraction of executions which only extracts information about self-modifications : execution is cut into paths without self-modification.

For the logical part, we tried to find a Curry-Howard correspondence between a language with self-modifications and logical world. For that we built an extension of λ-calculus with frozen terms, noted $\langle t \rangle$, that is, terms which cannot reduce. This terms are considered as data. Other terms are programs. We first prove that this language as expected properties like confluence. On the other hand, we found a type system where a subset of terms of this language can be expressed. Our type system is inspired by Linear Logic, without resources management. We prove that this system has good properties like subject reduction. We finally have extended the system with continuation and double negation. This extension can be expressed in a Krivine style, using a machine inspired by Krivine machine.

Keywords : malwares, self-modification, obfuscation, semantics, reflexion, linear logic