Université de Grenoble

Mémoire

pour obtenir

L'HABILITATION A DIRIGER DES RECHERCHES

présenté par

Frédéric PROST

CALCUL ET DEPENDANCE

Date de soutenance : 16 Novembre 2010

Jury :

Gilles Dowek	Professeur à l'Ecole Polytechnique
Jean-Claude Fernandez	Professeur à l'Université de Grenoble
Thomas Jensen	Directeur de recherche CNRS à l'IRISA Rennes
Hervé Martin	Professeur à l'Université de Grenoble
Alexandre Miquel	Maître de conférences habilité de l'Université Paris 7
Peter Selinger	Associate professor à Dalhousie University

Résumé

Le problème de savoir qui fait quoi pour qui est une question centrale dans toute organisation humaine. En programmation ce problème se décline sous l'angle des propriétés duales d'interférences ou de non-interférence : il s'agit de savoir quelles parties d'un programme sont dépendantes ou indépendantes les unes des autres. C'est à dire comment la modification de la valeur de certaines informations influe ou non sur la valeur d'autres informations.

Nous illustrerons nos recherches sur ce thème au travers de trois domaines d'applications différents : la sécurité, la programmation quantique et la réécriture de graphes.

1. La propriété de non-interférence est un support de base pour de confidentialité. L'objectif est de ne pas avoir d'interférence du domaine privé vers le domaine public. Nous étudierons ce problèe dans un cadre multiparadigme. Les analyses seront menées suivant différentes techniques comme le typage, la réécriture ou l'interprétation abstraite.
2. Nous étudierons la définition de lambda-calculs pour des calculs quantiques et d'une logique à la Hoare pour établir des jugements à propos de l'intrication.
3. Enfin, nous verrons comment la réécriture de graphes vue dans un formalisme catégorique peut être utilisée pour gérer l'espace mémoire et détecter le code qui n'est plus accessible (problème du ramasse-miettes).

Remerciements

Je tiens à remercier mon entourage pour la motivation et l'inspiration qu'il m'apporte. Je pense bien sûr aux chercheurs avec qui je travaille régulièrement notamment Rachid Echahed et Dominique Duval. L'environnement de travail dans mon équipe de recherche, CAPP, est également spécifique en ce qu'elle m'a permi de m'intéresser et de m'ouvrir à des problématiques que je ne connaissais pas du tout.

Je remercie également les relecteurs de ce mémoire qui ont permi d'améliorer plus que sensiblement la qualité d'icelui. Je voudrais particulièrement mettre en avant les discussions et la précision qui ont été apportés grâce aux remarques de Peter Selinger.

Table des matières

Chapitre 1

Introduction

Avant-propos

Le but du présent manuscript est de synthétiser et mettre en perspective notre travail de recherche depuis 2001, la date d'entrée en poste d'enseignant-chercheur à l'UJF et au laboratoire Leibniz, puis LIG, datant de novembre 2000.

Dans ce mémoire nous nous attachons principalement à la description et à la discussion des résultats obtenus. Ce document ne contient pas de preuves formelles et nous convions le lecteur aux travaux publiés pour d'avantage de précisions techniques. Ces travaux sont disponibles sur le web :

`http ://membres-lig.imag.fr/prost`

Nos travaux se regroupent autour d'un thème principal qui est celui de l'analyse des interférences (ou dualement de la non-interférence) au sein d'un programme. Nous traitons cette problématique dans trois domaines distincts :

- La confidentialité des données.
- L'informatique quantique.
- La transformation de graphes.

Pour chacun de ces trois domaines nous donnons une rapide introduction, les résultats obtenus et les problématiques particulières. Nous supposons simplement que le lecteur a une connaissance préalable des bases en informatique théorique. Nous indiquons des textes de références pour de plus amples informations.

Interférences en programmation

Le calcul des dépendances est, en informatique, historiquement lié aux processus d'optimisations dans les compilateurs [ALSU06]. Traditionnelle-

ment, dans les langages impératifs, on parle d'analyse de flot de données [Hec77] : le but est savoir comment certaines propriétés des données sont propagées le long des chemins d'éxécution d'un programme (le graphe de contrôle de flot). Cette analyse est à la base de nombreuses techniques d'optimisations : parallélisation, réordonnancement des calculs, propagation des constantes, élimination des sous-expressions communes etc.

Dans les langages fonctionnels l'analyse des dépendances trouve aussi des applications en optimisation : évaluation partielle [JGS93, CD93], analyse de "strictness" [Myc80], analyse de code mort [BB97, DP98, Pro00] (notamment en liaison avec l'extraction de contenu algorithmique de preuves comme c'est le cas dans Coq [BC04]) ou encore l'analyse dite de "program slicing" [BG96]. Typiquement ce que l'on cherche à découvrir sont les fonctions n'utilisant par leurs arguments. L'exemple le plus simple étant la fonction constante :

$$(\lambda x.\mathbf{0} \;\; p)$$

dans laquelle il est clair que p est du code mort. Bien sur les choses se compliquent à l'envie, si l'on considère simplement :

$$(\lambda x.(f \;\; x) \;\; p)$$

alors p sera du code mort si et seulement si f ne dépend pas de son argument. Ce que l'on cherche est précisément d'analyser ce type particulier de dépendances.

Notre approche est celle de l'analyse statique [NNH99], l'objectif est de déterminer des propriétés dynamiques (ici donc des propriétés d'interférences) sans pour autant éxécuter le programme correspondant. Il existe plusieurs écoles pour mener ces analyses. Nous nous concentrons principalement sur l'utilisation de système de typage et l'interprétation abstraite (nous utilisons également une approche basée sur les contraintes dans le cadre de l'analyse de consommation de mémoire définie en section 4.4.2). Les travaux que nous présentons sont, de ce point de vue, des extensions naturelles de [Pro00] et [Pro01] dans lesquels nous avons travaillé sur l'analyse statique dans le cadre du λ-calcul pur.

Nous avons élargi notre recherche en intégrant des aspects concurrentiels (voir en section 2.1) dans le cadre de la confidentialité. Là le but est d'éviter d'avoir des parties du code qui rendent publique des données secrètes. L'exemple typique étant donné par le programme suivant :

$$\mathsf{if}\ PIN{=}\ 0\ \mathsf{then}\ SPY := 0\ \mathsf{else}\ SPY := 1,$$

dans lequel la valeur de finale (publique) de la variable SPY va dépendre de la valeur (considérée secrète) de la variable PIN. La concurrence rajoute des difficultés en ce qu'une utilisation subtile de la synchronisation entre processus peut conduire à des pertes d'informations alors même que chaque processus pris individuellement est sûr.

Nous avons aussi travaillé sur la notion d'interférence même (voir en section 2.2.2). L'idée étant d'être moins strict et de permettre certaines interférences cela permettant de mieux refléter les situations que l'on rencontre dans la vie courante. Ainsi, lorsqu'on entre un code secret le système répond de manière publique le résultat d'un test d'une valeur privée. Il faut donc donner une définition de des interférences qui fasse qu'une telle interférence ne soit pas vue comme dangereuse.

Nous avons aussi étendu nos réflexions sur les interférences en considérant de nouveaux modèles de calculs notamment le quantique (dans lequel les notions d'interférences sont vues sous l'angle de l'enchevêtrement) ce que nous développons dans le chapitre 3. Ici l'objectif est de savoir quels sont les bits quantiques enchevêtrés (car agir sur un bit quantique peut avoir des effets sur tous ceux avec qui il est enchevêtré).

Nous avons enfin pris en compte la manipulation de pointeurs (modélisée par la réécriture de graphes) dans le chapitre 4 et les implications que cela entraine concernant la gestion de la mémoire. Il s'agit d'un type de dépendance qui se rapproche du code mort : nous cherchons à voir quels sont les parties d'un graphe qui deviennent inacessibles après réécriture.

Organisation du mémoire

Le mémoire est organisé autour de trois chapitres traitant chacun d'une problématique liée aux dépendances dans les calculs. Ces chapitres peuvent se lire de manière indépendante.

Dans le chapitre 2 nous présentons des résultats concernant le problème de la confidentialité vue sous l'angle de la non-interférence. Nous étendons les résultats connus dans un cadre multi-paradigme et nous montrons comment des techniques standard, basées sur le typage, peuvent s'étendre à un cadre avec concurrence. Enfin nous développons une notion d'interférences admissibles permettant de définir des politiques de sécurité plus souples que la non-interférence stricte.

Le chapitre 3 traite d'un type de dépendance nouveau en programmation avec l'apparition de modèles de calcul quantique. L'enchevêtrement de bits quantiques est un phénomène central des calculs quantiques, or deux bits quantiques enchevêtrés interfèrent : agir sur l'un (par une mesure ou une

porte unitaire) a potentiellement un effet sur l'autre. Cette situation n'est pas sans rappeler les problèmes liés à la notion d'alias quand on programme avec des pointeurs. Nous étudions, du point de vue de la programmation, comment manipuler (notamment parce que l'enchevêtrement casse les notions habituelles de portée dans les langages) et analyser la relation d'enchevêtrement (en proposant une logique à la Hoare pour décrire la relation d'enchevêtrement d'une mémoire quantique).

Enfin, dans le chapitre 4 nous abordons la problématique de l'analyse de l'utilisation d'une mémoire dans le cadre de la programmation de haut niveau avec pointeurs. Nous montrons comment une définition catégorique de la réécriture de graphes permet de caractériser le code qui n'est plus atteignable et ne peut plus interagir avec le programme, et qui donc peut être supprimé par ramasse miettes. Nous traitons également d'un formalisme catégorique de réécriture de graphes permettant au programmeur de manipuler explicitement la mémoire (alors que l'effacement d'une donnée est, dans ce formalisme, problématique) et aussi comment on peut se servir de ce formalisme pour baser une analyse de la quantité de mémoire requise par un programme.

Chapitre 2

Non-interférence et confidentialité

2.1 Le problème de la fuite d'information

2.1.1 Non-interférence et confidentialité

Le développement des télécomunications et l'informatisation toujours croissante de la société, on parle d'informatique ubiquitaire [Wei94] ou encore de cloud computing [Wei07], fait de la question de la confidentialité des données un défi majeur. Si auparavant ces questions existaient, elles concernaient principalement des activités particulières (armée, banque, etc.), aujourd'hui c'est tout un chacun qui est confronté à la gestion numérique de ses données : coordonnées banquaires, déclarations fiscales, carnet de santé, correspondance privée, réseaux sociaux etc. Il n'y a pratiquement pas un champs de l'existence qui échappe à ce mouvement double formé d'une part par une numérisation des informations et d'autre part par leur traitement, souvent automatique au moyen de programmes ou d'appareils communiquants entre eux, et leur transmission par les réseaux informatiques. Si la problématique peut se réduire à la simple question suivante : qui a le droit de savoir quoi ? Les réponses, voire l'interprétation de cette question, ne le sont pas.

Le traitement classique du contrôle d'accès aux données, initié par Goguen et Meseguer dans [GM82], repose d'une part sur la notion de non-interférence, et d'autre part sur le fait de classer les données en niveau de confidentialité (privée/publique pour faire simple mais on peut considérer un treillis pour représenter des niveaux de confidentialité plus subtils). L'idée est que deux parties d'un système n'interfèrent pas si ce qui est fait dans une partie n'a pas d'influence sur l'autre (et vice-versa). Ainsi, si une partie étiquetée publique ne dépent pas d'une autre partie étiqueté privée, il n'y a pas de fuite d'infor-

mation du privé vers le public. Le problème est que la notion "d'influences" sous-entendue dépend de ce qu'on entend par programmes aux comportements équivalents, ce qui n'est pas en soit un problème trivial dans un cadre non-déterministe [RS99].

Il y a eu un grand nombre de travaux dans ce domaine que l'on peut grouper en deux axes majeurs :

1. Le premier axe concerne les différents paradigmes de programmation étudiés : programmation impérative par exemple [SVI96], programmation fonctionelle cf. [Pro00], alègbres de processus cf. [HR00, AB02], combinaison de programmation fonctionelle et de processus communiquants [YH00], combinaison entre programmation impérative et processus communiquants [SV98a, BC02], systèmes multi-paradigmes [EPS03].
2. Le second axe consiste en l'étude de la notion même de non-interférence. On peut définir différentes notions de confinement de l'information : probabiliste, non déterministe, déterministe (voir [VS98, SM02]). Plusieurs adaptations de la non-interférences ont été proposées débouchant sur des notions comme la non-interférence approximative [PHW02b, PHW02a], des propositions d'affaiblissement de la non-interférence comme la déclassification [ZM01, GM04].

Nous présentons des travaux dans ces deux axes.

2.1.2 Non-interférence dans un cadre concurrent

L'analyse des interférences est notoirement plus compliquée en présence de processus concurrents. Une utilisation judicieuse de la synchronisation et des blocages permet des fuites d'informations assez subtiles. L'exemple minimal mettant en lumière cette difficulté est donné dans [BC02], il s'agit des trois processus, α, β, γ suivants (la syntaxe précise est décrite dans la section 2.2.1) :

$$\begin{array}{lcl} \alpha & \Leftarrow & [c_\alpha = \mathbf{0} \Rightarrow SPY := \mathbf{1}\,;\ c_\beta := \mathbf{0}];\ \theta \\ \beta & \Leftarrow & [c_\beta = \mathbf{0} \Rightarrow SPY := \mathbf{0}\,;\ c_\alpha := \mathbf{0}];\ \theta \\ \gamma & \Leftarrow & \big([PIN = \mathbf{1} \Rightarrow c_\alpha := \mathbf{0}]; \theta\big) + \\ & & \big([PIN = \mathbf{0} \Rightarrow c_\beta := \mathbf{0}]; \theta\big) \end{array}$$

dans lesquels PIN, SPY, c_α et c_β sont des constantes booléennes. c_α et c_β sont initialisées à $\mathbf{1}$. Une éxécution de $\alpha \parallel \beta \parallel \gamma$ va donc copier la valeur de PIN dans la constante publique SPY.

Bien que minimal cet exemple montre clairement les limitations des analyses de sécurité dans les langages impératifs comme celles de [SVI96] dans un contexte concurrent. Il s'agit d'une version simplifiée de l'exemple présenté

dans [SV98b]. On constate qu'on ne peut pas simplement se contenter d'appliquer une analyse à chacun des processus pris séparément : car isolément chacun de ces processus est sûr. En effet, d'une certaine manière c'est au moyen du flot de contrôle global que se produit la fuite d'information. Les processus α et β ne sont débloqués que quand la valeur de *PIN* est enregistrée dans c_α ou c_β par la mise à 0 de la bonne variable.

Nous présentons deux types d'approches pour tenter de cerner ce type de problème. Dans un premier temps, dans la section 2.2.1, nous développons une analyse basée sur une interprétation abstraite (cf. [CC77, Myc81, Hun91]) pour un cadre de programmation multiparadigme. Dans un second temps, dans la section 2.3 nous aborderons une approche basée sur le typage en utilisant une variante du calcul bleu de Boudol [Bou97] et en s'inspirant de travaux faits en programmation fonctionnelle [Pro00].

2.1.3 Interférences admissibles

D'un point de vue formel, la propriété de non-interférence, ou une version affaiblie permettant à l'information de progresser vers des niveaux de sécurité plus hauts, est très stricte. Il est même improbable qu'elle recouvre une situation pratique [RMMG01].

Un exemple typique dans lequel on utilise une fonction de déclassement est le mécanisme du mot de passe. Que l'on pense aux guichets automatiques, aux processus de login (sur un ordinateur, un site internet à accès restreint), aux codes PIN des téléphones portables, etc. la situation est schématiquement toujours la même : pour accéder à une zone, ou une information, secrète il faut fournir le bon mot de passe à un certain contrôleur. D'un point de vue de non-interférence stricte il y a une fuite d'information interdite dans ce type de situations. En effet le comportement du contrôleur qui est public change en fonction d'une donnée secrète. On peut même imaginer tapper tous les mots de passe possibles pour, au bout du compte, arriver à obtenir l'information confidentielle. Néanmoins de telles fuites d'informations sont acceptables. Premièrement parce que les fonctions permettants ces fuites sont *connues*, il n'est donc pas possible de directement divulguer un secret, il faut passer par certains points bien identifiés. Deuxièmement, comme ces fonctions sont identifiées, il est tout à fait possible de mettre au point une stratégie permettant d'empêcher les attaques en force : on peut par exemple figer le système (l'automate garde la carte banquaire) après trois tentatives d'identification infructueuses.

Un autre exemple de telles fuites d'informations est le cas de la cryptographie par clef publique. Si une donnée encodée est publique, comme la clef

de cryptage et l'algorithme sont connus, il est théoriquement possible d'encoder chaque message de la bonne longueur jusqu'à trouver celui qui produit le message codé. Il s'agit d'une variante à plus grande échelle du mécanisme de vérification des mots de passe dans les systèmes UNIX dans lesquels les mots de passes sont enregistrés encryptés et lisibles par tous.

Remarquons que ce schéma n'est pas lié directement à la cryptographie. Les fonctions de déclassifications jouent simplement le rôle d'interface entre les niveaux de confidentialité. Il n'est pas requis que ces fonctions soient d'une quelconque manière cryptographiques. Il est en dehors de notre étude de rechercher si une fonction de déclassification est "sûre" d'un point de vue pratique. Le problème auquel nous nous attelons est d'étudier la confidentialité par rapport à une certaine politique.

Nous cherchons à formaliser l'idée suivante : si une fonction est déclarée comme fonction de déclassement par la politique de confidentialité, alors nous faisons abstraction de la fuite d'information provenant de l'utilisation de cette fonction. En d'autres termes un processus est sûr si les seules fuites d'informations qu'il produit sont déclarées dans la politique de confidentialité.

Analyses statiques

Notre objectif est de développer des cadres théoriques permettant d'analyser ces propriétés de confidentialité de manière statique. Pour cela nous utilisons deux types d'approches. Une première approche, présenté en section 2.2 est celle de l'interprétation abstraite [CC77]. Dans notre cas cela revient à développer une sémantique opérationnelle abstraite (normalisante) qui va engendrer des contraintes qui, quand elles sont vérifiées par une certaine politique de sécurité, indiquent que le programme est sûr vis-à-vis de cette politique. Il est à noter que nous utilisons une approche tout à fait similaire pour la sémantique de la logique d'enchevêtrement développée en section 3.3.

La seconde approche que nous utilisons est celle du typage. L'idée est d'utiliser le typage 'standard' des langages de programmation comme un squelette sur lequel on pose des décorations. Nous montrons comment une telle approche peut être faite dans un cadre concurrent en section 2.3.

2.2 Analyse par interprétation abstraite

2.2.1 Non-interférence et programmation multiparadigme

Le modèle de calcul que nous utilisons dans cette section est une variante simplifiée de celui proposé dans [ES00, ES02] ; nous nous référons à ces travaux pour les définitions détaillées.

Si l'on se place à haut niveau, un programme, ou bien un composant, dans notre cadre consiste en la donnée de deux parties $C = (F, \Pi\mathcal{R})$. $\Pi\mathcal{R}$ est un ensemble de définitions de processus et F est un programme déclaratif, i.e., un ensemble de formules (ici nous considérons les systèmes de réécriture de termes (SRT)) que nous appellerons *store*. Nous supposons une certaine familiarité avec les SRT (voir par exemple [DJ90]).

Le modèle d'exécution d'un composant peut être schématisé de la façon suivante. Les processus communiquent par le biais d'un store commun F, c'est à dire en le modifiant de manière non montone. Par exemple par la redéfinition de constantes, en ajoutant ou retirant des formules de F. Une exécution de processus en parallèles va donc transformer le store F. Chaque modification du store est le résultat de l'éxécution d'une *action*.

Nous allons définir les différentes parties de ce modèle de calcul. La première étape est de formaliser la notion de store dans lequel les fonctions et les constantes sont définies.

Définition 1 *Un* store *est un* SRT *conditionnel multi-sorté* $F = \langle \Sigma, \mathcal{R} \rangle$*, composé d'une* signature Σ *et d'un ensemble de* règles *(ou de* formules*)* $\mathcal{R}$*. Une* signature $\Sigma = \langle S, \Omega \rangle$ *est un couple d'ensembles formé de* S *constitué de* sortes *et d'une famille d'opérateurs (indexés par* S*)* Ω*, encore appelés symboles de* fonction*, telle que* Σ *contient au moins la sorte* `Truth` *et le constructeur* True*. L'ensemble des* termes *correspondant à une signature* Σ *et aux variables* X *est dénoté* $T(\Sigma, X)$*. Les règles (les éléments de* $\mathcal{R}$*) sont de la forme* $l \to r \mid c$ *qui se lit : "*l *se réécrit en* r *si* c *est validé", où* l *et* r *sont termes de la même sorte, et* c *est un terme de sorte* `Truth`*.*

Nous supposons qu'il existe une fonction $eval(F, t)$, qui évalue le terme t à sa forme normale par rapport au store $F = \langle \Sigma, \mathcal{R} \rangle$.

On écrit $F \cup (l \to r \mid c)$ (resp. $F \setminus (l \to r \mid c)$) le store F dans lequel (resp. duquel) la règle $l \to r \mid c$ a été ajoutée (resp. retirée). De plus nous écrivons $F \bullet (l \to r \mid c)$ le store équivalent au store F dans lequel toutes les règles dont la partie gauche est l (i.e. les règles de la forme $l \to r' \mid c'$) ont été effacées et la règle $l \to r \mid c$ a été ajoutée.

Les actions permettent de modifier les stores. Nous distinguons deux types d'actions : *les actions élémentaires* comme l'affectation, l'ajout ou le retrait de règles, et *les actions gardées* qui sont éxécutées de façon atomique seulement dans le cas où leur garde s'évalue à True, ce qui rend possible une synchronisation de haut-niveau.

Définition 2 *Une* action α *est un couple constitué d'une garde* g *et d'une séquence d'actions élémentaires* a_i*, que l'on écrit :* $[g \Rightarrow \mathsf{a}_1; \ldots; \mathsf{a}_n]$*. Une* guarde *est un terme de sorte* Truth *dont la normalisation à* True *est décidable. Les* actions élémentaires a *que nous considérons sont l'affectation* (:=)*, l'ajout de règle dans un store* (tell)*, le retrait d'une règle d'un store* (del) *et l'action élémentaire qui ne fait rien* (skip)*.*

Les processus de base dans notre modèle de calcul sont, le processus qui termine avec succès θ, les actions gardées α, et les appels de processus $\mathrm{q}(t_1, \ldots, t_n)$. Comme il est habituel dans les algèbres de processus (consulter par exemple [Fok00]), nous considérons les opérateurs suivant pour combiner les processus : la composition parallèle ($\|$), séquentielle ($;$) et le choix non déterministe ($+$).

Définition 3 *Un* terme de processus p *est une expression définie par la grammaire suivante :*
$p ::= \theta \mid [g \Rightarrow \mathsf{a}] \mid p; p \mid p \parallel p \mid p + p \mid \mathrm{q}(t_1, \ldots, t_n)$.
Un processus q *est défini par une phrase de la forme suivante :*
$\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \sum_{i=1}^{m} \alpha_i ; p_i$ *où (pour chaque* i*)* α_i *est une action et* p_i *est un terme de processus, tel que les variables libres de* α_i *et* p_i *sont dans* $\{x_1; \ldots; x_n\}$.

Définition 4 *Un* système $\mathcal{S}$ *est un couple* $\langle F, \Pi\!R \rangle$*, où* F *est un store et* $\Pi\!R$ *est un ensemble de définitions de processus. Un* programme p *d'un système* $\mathcal{S}$ *est un processus clos* $\mathrm{q}(t_1, \ldots, t_n)$ *et tel que* $\mathrm{q}(x_1, \ldots, x_n)$ *est dans* $\Pi\!R$*.*

La sémantique opérationnelle intègre deux aspects orthogonaux : les utilisations successives du store pour l'évaluation de programmes déclaratifs et l'exécution des processus.

Les règles de transition sont données dans la figure Fig. 2.1 page 36. La relation de transition $\hookrightarrow$ décrit les modifications du store suite à l'éxécution d'une séquences d'actions élémentaires.

L'éxécution de processus est décrite par la relation de transition $\longrightarrow$. Les transitions sont de la forme $\langle F, p \rangle \longrightarrow \langle F', p' \rangle$ où F est un store et p est un terme de processus. La relation $\longrightarrow$ est définie modulo la relation d'équivalence classique selon laquelle les operateurs $\|$ et $+$ sont commutatifs et θ peut disparaitre.

Formalisation de la confidentialité

Nous allons maintenant préciser la notion de confidentialité. L'idée principale est d'assigner un *niveau de sécurité* à chacun des symboles de la signature. Un niveau de sécurité est un élément d'un treillis $\mathfrak{L}$ muni d'un ordre $\sqsubseteq$. Si π_1, π_2 sont deux éléments de $\mathfrak{L}$ et que $\pi_1 \sqsubseteq \pi_2$, on dit que π_2 est plus *confidentiel* que π_1. $\sqcup$ et $\sqcap$, désignent respectivement les opérateurs de majoration et minoration. Le plus grand élément de $\mathfrak{L}$ est $\top$ et son plus petit élément $\bot$.

Il faut maintenant associer à chaque symbole un niveau de confidentialité et un moyen d'inférer le niveau de confidentialité d'un terme en fonctions des symboles qui apparaissent dans ce terme. Cela se fait au moyen de la fonction de confidentialité dont la définition est la suivante :

Définition 5 (fonction de confidentialité) *On appelle fonction de confidentialité, ℓ, toute fonction des symboles de Ω vers $\mathfrak{L}$.*

ℓ s'étend naturellement aux termes de $T^{\mathcal{L}}(\Sigma, X)$, par :

$$\begin{array}{lcl} \ell(f(t_1, \ldots, t_n)) & = & (\bigsqcup_{i=1}^{n} \ell(t_i)) \sqcup \ell(f) \\ \ell(x) & = & \bot \end{array}$$

Il en est de même pour les règles de réécriture, la confidentialité de $l \to r \mid c$ est $\ell(l \to r \mid c)$ et est égale à $\ell(l)$. Enfin, on étend cette notion de confidentialité aux actions par :

$$\begin{array}{rcl} \ell([g \Rightarrow a_1; \ldots; a_n]) & = & \sqcap_{i=1}^{n} \ell(a_i) \\ \ell(\mathsf{tell}(l \to r \mid c)) = \ell(\mathsf{del}(l \to r \mid c)) & = & \ell(l) \\ \ell(\mathsf{skip}) & = & \bot \end{array}$$

Définition 6 (Règle de réécriture sûre) *Une règle de réécriture $l \to r \mid c$ est sûre quand les conditions suivantes sont vérifiées :*

$$(\ell(r) \sqsubseteq \ell(l)) \wedge (\ell(c) \sqsubseteq \ell(l))$$

Cette notion s'étend naturellement aux stores : un store est dit sûr si toutes les règles qu'il contient le sont.

Pour étendre cette notion de sûreté au processus il faut commencer par définir une notion d'équivalence entre store jusqu'à un certain niveau de confidentialité.

Définition 7 (Equivalence de stores) *Soient F_1, F_2 deux stores, et $\pi \in \mathfrak{L}$. On dit que F_1 et F_2 sont π-équivalents vis à vis de la fonction de confidentialité ℓ et on écrit $F_1 \equiv_{\pi}^{\ell} F_2$ ssi :*

1. $\forall p_1 = l_1 \to r_1 \mid c_1 \in \mathcal{R}_1$, *et* $\ell(p_1) \sqsubseteq \pi$ *à un renomage près,* $\exists p_2 = l_2 \to r_2 \mid c_2 \in \mathcal{R}_2$, *et* $p_1 = p_2$.
2. $\forall p_2 = l_2 \to r_2 \mid c_2 \in \mathcal{R}_2$, *et* $\ell(p_2) \sqsubseteq \pi$ *à un renomage de variables près,* $\exists p_1 = l_1 \to r_1 \mid c_1 \in \mathcal{R}_1$, *et* $p_2 = p_1$.

L'étape suivante est de définir une notion de bisimilarité entre processus jusqu'à un certain niveau de confidentialité. Deux processus p_1, p_2 seront dits bisimilaires au niveau π vis à vis de la fonction de confidentialité et ℓ ($\pi\ell$-bisimilaires) lorsqu'éxécutés sur des stores $\pi\ell$-équivalents, ils restent $\pi\ell$-bisimilaires. Informellement quand deux processus sont $\pi\ell$-bisimilaires cela signifie que leurs actions sont exactement les mêmes en dessous du niveau de confidentialité π. Si un des deux processus se bloque (du fait d'un test sur une valeur dont le niveau de confidentialité est supérieur à π) alors l'autre processus ne doit avoir que des actions de niveau supérieur à π (ce qui assure que les stores resteront π-équivalents).

Définition 8 ($\pi\ell$-bisimilarité) *Soient* p_1, p_2 *deux* $\Pi\mathcal{R}$ *termes de processus,* $\pi \in \mathfrak{L}$, *et* F_1, F_2 *deux stores, et* ℓ *une fonction de confidentialité telle que* $F_1 \equiv^{\ell}_{\pi} F_2$. *Une relation* $\approx^{\ell}_{\pi}$ *est une* $\pi\ell$*-bisimilarité si elle est symmétrique et si* $\langle F_1, p_1\rangle \approx^{\ell}_{\pi} \langle F_2, p_2\rangle$ *implique :*

- *Soit* p_1 *se réduit, et pour chaque* p_1' *tel que* $\langle F_1, p_1\rangle \longrightarrow \langle F_1', p_1'\rangle$, *alors il existe* p_2' *tel que* $\langle F_2, p_2\rangle \longrightarrow \langle F_2', p_2'\rangle$, *et* $F_1' \equiv^{\ell}_{\pi} F_2'$ *et* $\langle F_1', p_1'\rangle \approx^{\ell}_{\pi} \langle F_2', p_2'\rangle$.
- *Soit* p_1 *ne se réduit pas. Alors pour chaque* p_2' *tel que* $\langle F_2, p_2\rangle \longrightarrow \langle F_2', p_2'\rangle$, $F_1 \equiv^{\ell}_{\pi} F_2'$ *et* $\langle F_1, p_1\rangle \approx^{\ell}_{\pi} \langle F_2', p_2'\rangle$.

Lemme 1 *Soit* ℓ *une fonction de confidentialité,* π *un niveau de confidentialité,* p_0 *un terme de processus, et* F_0^1, F_0^2 *deux stores.* $\langle F_0^1, p_0\rangle \not\approx^{\ell}_{\pi} \langle F_0^2, p_0\rangle$ *implique l'existence d'un entier* N, *et de deux dérivations :*

$$\langle F_0^0, p_0\rangle \longrightarrow \langle F_1^0, p_1\rangle \longrightarrow \ldots \langle F_N^0, p_N\rangle$$
$$\langle F_0^1, p_0\rangle \longrightarrow \langle F_1^1, p_1\rangle \longrightarrow \ldots \langle F_N^1, p_N\rangle$$

telles que pour tout $j < N$, $F_j^0 \equiv^{\ell}_{\pi} F_j^1$ *et :*

1. *soit* $F_N^0 \not\equiv^{\ell}_{\pi} F_N^1$, *ou bien*
2. *il existe* $F^{\#}, p^{\#}$ *tel que pour tout* $j \in \{1, 2\}$ *:*
   - *soit* $\langle F_N^j, p_N\rangle \xrightarrow{a} \langle F_{N+1}^j, p_{N+1}\rangle \longrightarrow^* \langle F^{\#}, p^{\#}\rangle$ *et* $\langle F_N^{j-1}, p_N\rangle \not\xrightarrow{a}$,
   - *ou alors* $F_N^{j-1} \not\equiv^{\ell}_{\pi} F^{\#}$.

Définition 9 (Programme sûr) *Un programme* p *d'un système* $\langle F, \Pi\mathcal{R}\rangle$ *est dit* sûr *vis à vis d'une fonction de confidentialité* ℓ, *ssi pour tout* $\pi \in \mathfrak{L}$, *et pour tous* F_1, F_2 *tels que* $F_1 \equiv^{\ell}_{\pi} F_2 \equiv^{\ell}_{\pi} F$ *alors on a* $\langle F_1, \mathsf{p}\rangle \approx^{\ell}_{\pi} \langle F_2, \mathsf{p}\rangle$.

Représentation abstraite

Pour analyser si un programme est sûr nous développons une sémantique opérationnelle abstraite, qui termine tout le temps, dont le but est d'engendrer un ensemble de contraintes qui sont des inéquations entre niveaux de confidentialité. Si ces contraintes sont satisfaites alors le programme analysé est prouvé sûr au sens de la définition 9. Dans le cas contraire il se peut que le programme ne soit pas sûr (notre analyse est donc une approximation conservative).

L'idée principale est la suivante : nous utilisons un squelette d'éxécution pour l'analyse, pour cela, au cours de l'éxécution abstraite, nous collectons juste les noms des symboles utilisés et nous abstrayons le store à un ensemble d'inéquations. Ce squelette d'éxécution est une version opérationnelle des annotations que l'on peut poser sur les types pour ce genre d'analyse, technique que nous étudierons plus précisément dans la section 2.3.

Sémantique opérationnelle abstraite Un store abstrait est un ensemble d'inéquations sur des variables et des éléments de $\mathfrak{L}$. Pour chaque symbole de la signature de F, nous introduisons une constante dénotant son niveau de confidentialité. La sémantique abstraite va collecter les inéquations au fur et à mesure de l'éxécution abstraite (dont la définition même assure la terminaison). L'idée est de produire l'ensemble des inéquations correspondant à un programme. Ensuite si une fonction de confidentialité ℓ définie sur $\mathcal{S}$ est telle que l'ensemble d'inéquations est vérifié, alors on assure que le programme est sûr vis-à-vis de ℓ.

Définition 10 (Inéquations de confidentialité et formules) *Les formules de confidentialité* $\mathfrak{f}$ *sont définies par :*

$$\mathfrak{f} ::= \pi \mid x \mid \mathfrak{f} \sqcap \mathfrak{f} \mid \mathfrak{f} \sqcup \mathfrak{f}$$

où x *dénote des constantes. Les* inéquations de confidentialité *sont des termes de la forme suivante :*

$$\mathfrak{f}_1 \sqsubseteq \mathfrak{f}_2$$

Définition 11 (Stores abstraits) *Un store abstrait* $F^{\mathcal{A}}$ *est un couple formé d'une signature abstraite et d'un ensemble abstrait de règles* $\langle \Sigma^{\mathcal{A}}, \mathcal{R}^{\mathcal{A}} \rangle$*, tel que :*

- $\Sigma^{\mathcal{A}}$ *est un ensemble de constantes.*
- $\mathcal{R}^{\mathcal{A}}$ *est un ensemble d'inéquations construit avec des symboles appartenant à* $\Sigma^{\mathcal{A}}$*.*

Soit $F = \langle \Sigma, \mathcal{R} \rangle$ un store, on définit le store abstrait, $F^{\mathcal{A}} = \langle \Sigma^{\mathcal{A}}, \mathcal{R}^{\mathcal{A}} \rangle$ de F par :

- *Pour tout élément $f \in \Sigma$, il y a un symbole x_f dans $\Sigma^{\mathcal{A}}$.*
- *Pour toutes les règles $l \to r \mid c$ de $\mathcal{R}$, il y a des règles $\ell^{\mathcal{A}}(r) \sqsubseteq \ell^{\mathcal{A}}(l), \ell^{\mathcal{A}}(c) \sqsubseteq \ell^{\mathcal{A}}(l)$ dans $\mathcal{R}^{\mathcal{A}}_{\ell}$, où $\ell^{\mathcal{A}}$ est la fonction inductivement définie par :*

$$\ell^{\mathcal{A}}(f) = x_f, \forall f \in \Sigma$$
$$\ell^{\mathcal{A}}(f(t_1, \ldots, t_n)) = \ell^{\mathcal{A}}(f) \sqcup \ell^{\mathcal{A}}(t_1) \sqcup, \ldots, \sqcup \ell^{\mathcal{A}}(t_n)$$

Dans la suite nous noterons $t^{\mathcal{A}}$ pour simplifier l'écriture de $\ell^{\mathcal{A}}(t)$.

Le but du store abstrait est de collecter les contraintes qui doivent être vérifiées par un programme sûr. Ces contraintes sont engendrées par une sémantique opérationnelle abstraite définie sur les stores abstraits et les processus abstraits (contrairement aux stores et processus "standard"). De plus on ajoute une information σ de $\mathfrak{L}$. En cela nous suivons une idée développée dans [BC01] : σ dénote le plus haut niveau de confidentialité des gardes consultées jusqu'à un certain point dans le programme. Une éxécution abstraite va engendrer des contraintes qu'elle enregistre dans le store abstrait. Ces contraintes dépendent de σ tout comme du niveau de confidentialité des termes manipulés par le programme.

La sémantique opérationnelle abstraite des action modifie un store abstrait $F^{\mathcal{A}}$, ainsi qu'un niveau de confidentialité σ de $\mathfrak{L}$. Elle est défine comme suit :

$$\langle F^{\mathcal{A}}, c := t; a, \sigma \rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}} \wedge (t^{\mathcal{A}} \sqsubseteq c^{\mathcal{A}}) \wedge (\sigma \sqsubseteq c^{\mathcal{A}}), a, \sigma \rangle \qquad (\mathsf{A}ea_{:=})$$

$$\langle F^{\mathcal{A}}, \mathsf{tell}(l \to r \mid c); a, \sigma \rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}} \wedge (r^{\mathcal{A}} \sqsubseteq l^{\mathcal{A}}) \wedge (c^{\mathcal{A}} \sqsubseteq l^{\mathcal{A}}) \wedge (\sigma \sqsubseteq l^{\mathcal{A}}), a, \sigma \rangle \qquad (\mathsf{A}ea_{\mathsf{tell}})$$

$$\langle F^{\mathcal{A}}, \mathsf{del}(l \to r \mid c); a, \sigma \rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}} \wedge (\sigma \sqsubseteq l^{\mathcal{A}}), a, \sigma \rangle \qquad (\mathsf{A}ea_{\mathsf{del}})$$

$$\langle F^{\mathcal{A}}, \mathsf{skip}; a, \sigma \rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}}, a, \sigma \rangle \qquad (\mathsf{A}ea_{\mathsf{skip};})$$

La relation de transition abstraite sur les termes de processus à la forme suivante :

$$\langle F^{\mathcal{A}}, \mathcal{M} \rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}' \rangle$$

où $\mathcal{M}$ est défini par la grammaire suivante :

$$\mathcal{M} ::= \langle p, \sigma \rangle \mid \mathcal{M} \parallel^{\mathcal{A}} \mathcal{M} \mid \mathcal{M} +^{\mathcal{A}} \mathcal{M} \mid \mathcal{M} ;^{\mathcal{A}} \mathcal{M}$$

L'introduction d'opérateurs abstraits ($+^{\mathcal{A}}$, $\parallel^{\mathcal{A}}$, $;^{\mathcal{A}}$) est nécessaire pour la définition de la transition abstraite car nous considérons les couples formés

par un processus et un niveau de confidentialité (σ) indiquant le plus haut niveau de confidentialité rencontré. L'opérateur abstrait le plus important est $\|^{\mathcal{A}}$, les autres sont définis pour des raisons de cohérence de notation mais n'ont pas d'influence sur l'analyse. $\|^{\mathcal{A}}$ est requis parce qu'il est possible que dans un processus de type $p \parallel q$, le processus p se serve d'une garde d'un haut niveau alors que le processus q ne travaille que sur des niveaux de confidentialité faibles. Si on ne dupplique pas σ, alors ce type de processus serait analysé comme non sûr car les contraintes engendrées par p pourraient se révéler trop restrictives pour q. D'un certain point de vue, le rôle de $\|^{\mathcal{A}}$ est similaire au rôle de la règle de sous-typage dans les analyses basées sur le typage (par exemple dans [BC01]). En effet dans les analyses basées sur le typage, il est possible de construire des contraintes différentes (donc d'associer des types différents) pour les deux parties d'un processus parallèle et ensuite d'utiliser la règle de sous-typage pour unifier le type des deux processus dans l'arbre de typage.

Nous définissons une fonction ϕ des termes de $\mathcal{M}$ aux termes de processus, le résultat est le terme original correspondant à l'expression $\mathcal{M}$.

$$\begin{array}{ll} \phi(\langle p, \sigma\rangle) = p & \phi(\mathcal{M} \parallel^{\mathcal{A}} \mathcal{M}') = \phi(\mathcal{M}) \parallel \phi(\mathcal{M}') \\ \phi(\mathcal{M} +^{\mathcal{A}} \mathcal{M}') = \phi(\mathcal{M}) + \phi(\mathcal{M}') & \phi(\mathcal{M} ;^{\mathcal{A}} \mathcal{M}') = \phi(\mathcal{M}) ; \phi(\mathcal{M}') \end{array}$$

Nous définissons $\equiv^{\mathcal{A}}$ une congruence structurelle sur les termes de $\mathcal{M}$:

$$\frac{p_1 \equiv_p p_2}{\langle p_1, \sigma\rangle \equiv^{\mathcal{A}} \langle p_2, \sigma\rangle} \quad (\mathsf{A}Eq_{\equiv_p})$$

$$\mathcal{M}_1 \parallel^{\mathcal{A}} \mathcal{M}_2 \equiv^{\mathcal{A}} \mathcal{M}_2 \parallel^{\mathcal{A}} \mathcal{M}_1 \quad (\mathsf{A}Eq_{\parallel^{\mathcal{A}}})$$

$$\mathcal{M}_1 +^{\mathcal{A}} \mathcal{M}_2 \equiv^{\mathcal{A}} \mathcal{M}_2 +^{\mathcal{A}} \mathcal{M}_1 \quad (\mathsf{A}Eq_{+^{\mathcal{A}}})$$

Cette congruence permet de gérer les opérateurs abstraits et va être utilisée pour définir une sémantique opérationelle abstraite des termes de processus.

La sémantique opérationnelle abstraite des termes de processus est définie sur les couples de la forme $\langle F^{\mathcal{A}}, \mathcal{M}\rangle$. Si $\mathcal{M}$ est de la forme $\langle p_1 \parallel p_2, \langle \sigma, \tau\rangle\rangle$, il est nécessaire de "traduire" l'opérateur $\parallel$ en $\|^{\mathcal{A}}$. En effet, c'est par l'utilisation de l'opérateur $\|^{\mathcal{A}}$ que l'on peut collecter les contraintes engendrées par l'éxécution de p_1 et p_2. Symétriquement, quand un processus d'une composition parallèle s'arrête il faut réunir les résultats calculés par les deux parties. Pour cela nous définissons une relation de transformation $\longmapsto$ qui permet d'introduire et d'éliminer $\|^{\mathcal{A}}$.

$$\langle p_1 \parallel p_2, \sigma\rangle \longmapsto \langle p_1, \sigma\rangle \parallel^{\mathcal{A}} \langle p_2, \sigma\rangle \qquad (\mathsf{A} \parallel^{\mathcal{A}} -I)$$

$$\langle p_1 + p_2, \sigma\rangle \longmapsto \langle p_1, \sigma\rangle +^{\mathcal{A}} \langle p_2, \sigma, \rangle \qquad (\mathsf{A} +^{\mathcal{A}} -I)$$

$$\langle p_1; p_2, \sigma\rangle \longmapsto \langle p_1, \sigma\rangle ;^{\mathcal{A}} \langle p_2, \sigma, \rangle \qquad (\mathsf{A} ;^{\mathcal{A}} -I)$$

$$\langle \theta, \sigma_1\rangle \parallel^{\mathcal{A}} \langle \theta, \sigma_2\rangle \longmapsto \langle \theta, \sigma_1 \sqcup \sigma_2, \rangle \qquad (\mathsf{A} \parallel^{\mathcal{A}} -E)$$

$$\langle \theta, \sigma_1\rangle ;^{\mathcal{A}} \langle p_2, \sigma_2\rangle \longmapsto \langle p_2, \sigma_1 \sqcup \sigma_2, \rangle \qquad (\mathsf{A} ;^{\mathcal{A}} -E)$$

Il est clair que $\longmapsto$ est confluente et fortement normalisante. On écrit $\overline{\mathcal{M}}^{\mathsf{nf}}$ la $\longmapsto$ forme normale de $\mathcal{M}$. L'idée est remplacer chaque $\parallel$ par un $\parallel^{\mathcal{A}}$ pour calculer les contraintes engendrées par les processus en parallèle de manière séparée (règle $\mathsf{A} \parallel^{\mathcal{A}} -I$) et symétriquement de pouvoir fusionner les résultats quand un processus termine son éxécution (rule $\mathsf{A} ;^{\mathcal{A}} -E$).

Les règles de la sémantique opérationnelle abstraite sont définies par :

Définition 12 (Sémantique opérationelle abstraite)

$$\frac{\overline{\mathcal{M}_1}^{\mathsf{nf}} \equiv^{\mathcal{A}} \mathcal{M}_2 \quad \langle F^{\mathcal{A}}, \mathcal{M}_2\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_3\rangle \quad \overline{\mathcal{M}_3}^{\mathsf{nf}} \equiv^{\mathcal{A}} \mathcal{M}_4}{\langle F^{\mathcal{A}}, \mathcal{M}_1\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_4\rangle} \qquad (\mathsf{A}P_{\equiv^{\mathcal{A}}})$$

$$\frac{(\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \Sigma_{j=1}^{m} \alpha_j \,; p_j) \in \Pi\!\!R \quad \langle F^{\mathcal{A}}, \langle (\Sigma_{j=1}^{m} \alpha_j \,; p_j)[x_i/t_i], \sigma\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}'\rangle}{\langle F^{\mathcal{A}}, \langle \mathrm{q}(t_1, \ldots, t_n), \sigma\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}'\rangle} \qquad (\mathsf{A}P_{abs})$$

$$\frac{\langle F^{\mathcal{A}}, \langle a_1; \ldots; a_n; \mathsf{skip}, \sigma \sqcup g^{\mathcal{A}}\rangle\rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \langle \mathsf{skip}, \sigma \sqcup g^{\mathcal{A}}\rangle\rangle}{\langle F^{\mathcal{A}}, \langle [g \Rightarrow a_1; \ldots; a_n], \sigma\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \langle \theta, \sigma \sqcup g^{\mathcal{A}}\rangle\rangle} \qquad (\mathsf{A}P_{guard})$$

$$\frac{\langle F^{\mathcal{A}}, \langle p_1, \sigma\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \langle p_1', \sigma'\rangle\rangle}{\langle F^{\mathcal{A}}, \langle p_1; \ldots; p_n, \sigma'\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \langle p_1'; \ldots; p_n, \sigma'\rangle\rangle} \qquad (\mathsf{A}P_{;})$$

$$\frac{\langle F^{\mathcal{A}}, \mathcal{M}_1\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_1'\rangle}{\langle F^{\mathcal{A}}, \mathcal{M}_1 \parallel^{\mathcal{A}} \mathcal{M}_2\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_1' \parallel^{\mathcal{A}} \mathcal{M}_2\rangle} \qquad (\mathsf{A}P_{\parallel^{\mathcal{A}}})$$

$$\frac{\langle F^{\mathcal{A}}, \langle p_1, \sigma\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}\rangle}{\langle F^{\mathcal{A}}, \langle p_1 + p_2, \sigma\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}\rangle} \qquad (\mathsf{A}P_{+})$$

Lemme 2 *Soit $\mathcal{S} = \langle F, \Pi\!\!R\rangle$, et p un terme de processus de $\mathcal{S}$. Si $\langle F, p\rangle \longrightarrow \langle F', p'\rangle$ alors pour tout $\mathcal{M}$ tel que $\phi(\mathcal{M}) = p$ et pour chaque ensemble de contraintes C, on a $\langle F^{\mathcal{A}} \cup C, \mathcal{M}\rangle \longrightarrow^{\mathcal{A}} \langle F'^{\mathcal{A}} \cup C', \mathcal{M}'\rangle$ avec $\phi(\mathcal{M}') = p'$ et $C \subseteq C'$.*

Analyse de programme

Pour analyser un programme l'idée est de faire toute les éxécutions abstraites possibles d'un programme et de collecter les inéquations calculées au cours de ce calcul. S'il existe une substitution des variables des formules de confidentialité vers $\mathfrak{L}$ satisfaisant les inéquations alors le programme analysé est sûr vis-à-vis de la fonction de confidentialité engendrée par cette substitution, c'est à dire une fonction de confidentialité qui assigne à f le même niveau que celui assigné à x_f (cf. définition 11).

Ce qui rend le calcul possible est qu'il n'y pas une infinité d'éxécutions abstraites possibles. Plus précisément, il arrive un moment où on ne peut plus engendrer de nouvelles inéquations de confidentialité. Cela provient du fait que les actions élémentaires abstraites ne modifient pas les valeurs du store (cf. règles $\mathsf{A}ea_{:=}$, $\mathsf{A}ea_{\mathsf{tell}}$, $\mathsf{A}ea_{\mathsf{del}}$) mais augmentent le nombre des inégalités de confidentialité. Comme le nombre de symboles dans un programme est fini, et que l'éxécution abstraite est purement symbolique, ce nombre est fini. Il est donc théoriquement possible de considérer l'arbre d'éxécution du programme (les branches infinies étants coupées quand on ne peut plus récupérer de nouvelle information). L'union de tous les stores abstraits que l'on récupère aux feuilles de cette éxécution abstraite est le résultat de l'analyse.

Définition 13 (Réduction d'analyse) *La réduction d'analyse $\rightsquigarrow$ est une relation entre triplets de la forme $\langle F^{\mathcal{A}}, \mathcal{M}, \mathfrak{H}\rangle$, où $\mathfrak{H}$ est un ensemble de couples de la forme $\langle \mathrm{q}, [\ell^{\mathcal{A}}(t_1); \ldots ; \ell^{\mathcal{A}}(t_n)]\rangle$. Elle est définie comme suit :*

- *Si $\langle F^{\mathcal{A}}, \mathcal{M}\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}'\rangle$ en utilisant une règle de réduction différente de $\mathsf{A}P_{abs}$, alors*
$$\langle F^{\mathcal{A}}, \mathcal{M}, \mathfrak{H}\rangle \rightsquigarrow \langle F^{\mathcal{A}\prime}, \mathcal{M}, \mathfrak{H}\rangle$$
- *Si $\mathcal{M} = \langle \mathrm{q}(t_1, \ldots, t_n) \Leftarrow \Sigma_{j=1}^{m} \alpha_j \,; p_j, \sigma\rangle$ et $\langle F^{\mathcal{A}}, \langle (\Sigma_{j=1}^{m} \alpha_j \,; p_j)[x_i/t_i], \sigma\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}'\rangle$, alors*
$$\langle F^{\mathcal{A}}, \mathcal{M}, \mathfrak{H}\rangle \rightsquigarrow \begin{cases} \langle F^{\mathcal{A}}, \langle \theta, \sigma\rangle, \mathfrak{H}\rangle \text{ si } \langle \mathrm{q}, [t_1^{\mathcal{A}}; \ldots ; t_n^{\mathcal{A}}]\rangle \in \mathfrak{H} \\ \langle F^{\mathcal{A}\prime}, \mathcal{M}', \mathfrak{H} \cup \langle \mathrm{q}, [t_1^{\mathcal{A}}; \ldots ; t_n^{\mathcal{A}}]\rangle\rangle \text{ sinon} \end{cases}$$

Comme la sémantique opérationnelle abstraite est purement symbolique, et n'utilise que des termes abstraits ($t^{\mathcal{A}}$) à la place de termes (t), le nombre de combinaison d'appels de processus est fini, donc il n'y a pas de séquence de réduction $\rightsquigarrow$ infinie.

Théorème 1 *La relation $\rightsquigarrow$ est fortement normalisante.*

Nous définissons maintenant une notion de squelette de programme. Informellement, il s'agit de la collection de tous les stores abstraits calculés par les éxécutions abstraites.

Définition 14 (Squelette de programme) *Soit* p *un programme d'un système* $\mathcal{S} = \langle F, \Pi R \rangle$. *On appelle* squelette *d'un programme* p, *et on écrit* $\mathsf{p}^{\natural}_{F}$, *l'ensemble* $\{F_i^{\mathcal{A}}\}_{i \in I}$ *avec* I *un ensemble d'indice tels que,* i *est dans* I *si* $\langle F_i^{\mathcal{A}}, \theta, \sigma_i \rangle$ *est atteignable à partir de* $\langle F^{\mathcal{A}}, \mathsf{p}, \bot \rangle$ *en utilisant la relation* $\rightsquigarrow$.

Définition 15 (Ensemble de contraintes et satisfaisabilité) *Un ensemble de contraintes* $F^{\mathcal{A}}$ *est* satisfaisable *par une substitution* $\mathfrak{S}$ *de* $\Sigma^{\mathcal{A}}$ *vers* $\mathfrak{L}$, *ssi pour chaque règle* $l \sqsubseteq r$ *de* $\mathcal{R}^{\mathcal{A}}$, *alors* $\mathfrak{S}(l) \sqsubseteq \mathfrak{S}(r)$ *est correct, où* $\mathfrak{S}(l)$ *est l'extension naturelle de* $\mathfrak{S}$ *aux termes de la forme* $x_1 \sqcup \ldots \sqcup x_n$.

Définition 16 (Compatibilité des store) *Un store abstrait* $F^{\mathcal{A}}$ *est dit* compatible *avec une fonction de confidentialité* ℓ *ssi* $\mathfrak{S}_\ell$ *satisfait* $F^{\mathcal{A}}$. *Où* $\mathfrak{S}_\ell$ *est défini par :*

$$\forall f \in F. \mathfrak{S}_\ell(x_f) = \ell(f)$$

Un programme sera sûr relativement à une fonction de confidentialité si son squelette est compatible avec cette fonction de confidentialité. L'inverse n'étant pas vrai, il existe des programmes sûrs compatibles qui sont catalogués comme non sûrs. Typiquement :

$$\mathrm{q}(x) \Leftarrow \begin{array}{l} \\ + \\ \\ \end{array} \begin{array}{l} [PIN = 12 \Rightarrow SPY := 0; \mathsf{skip}]\,;\theta \\ \\ [PIN \neq 12 \Rightarrow SPY := 0; \mathsf{skip}]\,;\theta \end{array}$$

est analysé comme non sûr relativement à une fonction de confidentialité qui assigne les niveaux $\top$ à PIN, et $\bot$ à SPY, alors qu'en fait la valeur finale de SPY ne dépend pas de la valeur de PIN, et il n'y a pas de fuite d'information dans ce programme.

Le fait que si une fonction de confidentialité satisfait un squelette de programme alors le programme considéré est sûr provient intuitivement du fait suivant : un processus est sûr s'il est bisimilaire à lui même relativement à chaque niveau de confidentialité. Donc par contradiction si un processus n'est pas sûr il existe un niveau de confidentialité π tel que la bisimilarité ne tienne pas. Il y a deux cas possibles : soit une modification observable entre les valeurs de niveau inférieur à π apparaît. Nous devons montrer que ce n'est pas possible si la fonction de confidentialité ℓ est compatible avec $\mathsf{p}^{\natural}_{F}$. C'est le cas car soit toutes les gardes sont inférieures à π donc l'éxécution abstraite assure que le niveau des actions élémentaires effectuées est inférieur à π et dans ce cas elles devraient donner le même résultat, ou bien une garde est supérieure à π mais dans ce cas les actions élémentaires modifient le store dans une zone non observable car d'un niveau supérieur à π.

Théorème 2 (Sûreté) *Soit* $\mathcal{S} = \langle F, \Pi\!R \rangle$ *un système,* p *un programme sur* $\mathcal{S}$*,* ℓ *une fonction de confidentialite sur* F*,* $\mathsf{p}^\natural_F$ *le squelette de* p*, alors si* $\mathsf{p}^\natural_F$ *est compatible avec* ℓ *alors* p *est sûr vis-à-vis de* ℓ*.*

2.2.2 Politique de confidentialité

Notre objectif est ici de mener une analyse de confidentialité pour laquelle l'utilisateur peut définir ce qu'il considère comme des interférences sûres. Nous commençons par décrire comment on peut définir une telle politique de confidentialité et ensuite comment on peut adapater l'analyse que nous venons de présenter pour en tenir compte.

Une politique de confidentialité est modélisée au moyen d'un système de réécriture confluent et normalisant. Les formes normales d'une telle politique de confidentialité sont les éléments du treillis de confidentialité, ce sont les niveaux de confidentialité des données. Pour chaque fonction f du store on associe un ensemble de règles de réécriture de la forme $f(\pi 1, x) \rightarrow \pi_2$, avec π_1, et π_2 des éléments du treillis de confidentialité. Dans une première approximation on peut voir ces règles de réécriture comme les profils de sécurité des fonctions. Ces règles formant la politique de confidentialité modélisent les hypothèses faites par le programmeur concernant la sûreté des fonctions. Par exemple il semble normal de déclarer que le résultat d'une fonction d'encryptage est public même si la fonction dépend de données privées. Cependant, il est clair qu'on peut définir une politique de sécurité avec la règle suivante : $id(x) \rightarrow \bot$, où id est la fonction identité. Dans ce cas la politique de sécurité est maladroite car $id(d)$, est évalué comme ayant un niveau de confidentialité le plus bas possible, permet de rendre public n'importe quelle information d. Le fait qu'une politique de sécurité soit mal définie est en dehors du champs de notre étude.

Ainsi pour chaque fonction f nous considérons deux systèmes de réécritures : le premier est le classique, celui permettant de calculer sur les données (typiquement $id(x) \rightarrow x$) et le second définissant la politique associée à cette fonction (par exemple $id(\top) \rightarrow \bot, id(\bot) \rightarrow \top$)

L'objectif est donc de définir une notion de processus sûrs relativement à une politique de sécurité. Il faut pour cela être en mesure de faire la distinction entre les interférences autorisées et celles interdites (les interférences classiques). C'est ce que permet de faire la *politique de sécurité* : grossièrement cette politique de sécurité va remplacer la fonction de confidentialité par un système de réécriture ayant des propriétés spéciales qui font que pour tout terme clos sa forme normale sera dans $\mathfrak{L}$. Ce système de réécriture est une manière d'encoder la fonction ℓ, qui donne plus de souplesse (le niveau de confidentialité d'un terme n'étant pas forcément la borne supérieure du

niveau de ses sous-termes mais pouvant résulter d'un calcul) : on parle de fonction de déclassement, ou de terme déclassé, quand le niveau de sécurité d'un terme est plus faible que celui d'un de ces sous-termes.

Pour analyser la sûreté des programmes dans un tel contexte nous utilisons une méthode semblable à celle définie en 2.2.1 mais en utilisant une évaluation de processus spécifique dans laquelle les termes déclassés sont évalués sur un store commun, rendant l'évaluation d'un terme déclassé égale sur deux stores différents. Il faut également développer une notion d'équivalence de processus qui soit en ligne avec cette évaluation. Il est alors possible d'avoir un résultat standard selon lequel les informations ne peuvent aller que vers des niveaux de confidentialité plus élevés.

On pourra noter que cette même approche, une sémantique collectant tous les résultats abstraits et une sémantique opérationnelle simulant des valeurs (ici communes pour deux stores différents et en ce qui concerne le quantique donnant un résultat non déterministe pour la mesure), est utilisée pour l'analyse d'intrication séparabilité, notamment la définition de l'état quantique abstrait, cf. section 3.3.3.

Définition de la politique de sécurité

Nous utilisons les notions usuelles de réécritures de termes (voir [DP01] pour plus de détails). Nous notons $t!_R$ la forme normale de t selon le système de réécriture R. Nous utilisons la notion de descendants, qui est utilisée pour prendre en compte les déclassifications, dont nous rappelons la définition.

Définition 17 *Soit $A = t \to_{u,l\to r} t'$ un pas de réécriture pour la réduction d'un terme t en t' à la position u en utilisant la règle $l \to r$. L'ensemble des* descendants *(ou des* résiduels*) d'une position v par A, notée $v \setminus A$, est*

$$v \setminus A = \begin{cases} \emptyset & \text{si } v = u \cdot p \text{ et} \\ & l|_p \text{ n'est pas une variable,} \\ \{v\} & \text{si } u \not\leq v, \\ \{u \cdot p' \cdot q \mid r|_{p'} = x\} & \text{si } v = u \cdot p \cdot q \text{ et} \\ & l|_p = x, \text{ et } x \text{ est une variable.} \end{cases}$$

L'ensemble des descendants *d'une position par une séquence de réduction B est défini par induction comme suit :*

$$v \setminus B = \begin{cases} \{v\} & \text{si } B \text{ est la dérivation vide,} \\ \bigcup_{w \in v \setminus B'} w \setminus B'' & \text{si } B = B'B'', \text{ où } B' \text{ est} \\ & \text{le premier pas de réécriture de } B. \end{cases}$$

Une position identifiant de manière univoque un sous-terme d'un terme, la notion de *descendant* pour les termes découle directement de la notion de descendance pour les positions.

Définition 18 (Politique de confidentialité) *Une politique de confidentialité, $\mathcal{SP}$, est un système de réécriture de terme confluent et normalisant défini sur la signature $\Sigma \cup \mathfrak{L}$ telle que :*

- *$\mathcal{SP}$ n'introduit rien par rapport à $\mathfrak{L}$. C'est à dire que pour tout terme clos, t, sur $\Sigma \cup \mathfrak{L}$, $t!_{\mathcal{SP}}$ est dans $\mathfrak{L}$.*
- *$\mathcal{SP}$ n'introduit pas de confusion sur $\mathfrak{L}$. c'est à dire que $\forall \tau_1, \tau_2 \in \mathfrak{L}, \tau_1 \neq \tau_2 \implies \tau_1 \not\xrightarrow{*}_{SP} \tau_2$*
- *Les fonctions définies dans Σ sont monotones par rapport au niveau de confidentialité, c'est à dire que $\forall \tau_1, \ldots, \tau_n \in \mathfrak{L}, \forall \tau_i' \in \mathfrak{L}, \tau_i \sqsubseteq \tau_i' \implies f(\tau_1, \ldots, \tau_i, \ldots, \tau_n)!_{\mathcal{SP}} \sqsubseteq f(\tau_1, \ldots, \tau_i', \ldots, \tau_n)!_{\mathcal{SP}})$*

La condition de non confusion garanti que deux niveaux de confidentialité différents ne seront pas rendus égaux par la politique de sécurité. La dernière condition n'est pas obligatoire mais apparaît pertinente dans un contexte de contrôle de la fuite d'information.

Exemple 1 *Soit $\Sigma = \{f, g, h\}$ une signature consistant en trois symboles de fonction avec les arités respectives 2, 2 et 1. Soit $\mathfrak{L} = \{\bot, \top\}$ et $\bot \sqsubseteq \top$. On peut définir une politique de confidentialité $\mathcal{SP}$,au moyen des règles suivantes :*

$$f(x, y) \to \top \quad g(x) \to x$$
$$h(\top, x) \to \bot \quad h(\bot, x) \to \bot$$

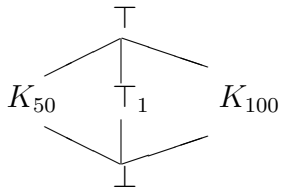
$g(f(\top, \bot))$ a pour forme normale $\top$ alors que $g(h(\top, \top))$ a pour forme normale $\bot$.

On peut noter que contrairement à l'usage le niveau de confidentialité peut diminuer au cours du calcul. Dans $\mathcal{SP}$, la fonction h peut déclasser de l'information. En effet, la règle $h(\top, x) \to \bot$ dépend d'une valeur de niveau de confidentialité $\top$ mais le niveau du résultat est $\bot$. Cela signifie que cette politique de sécurité autorise *la fonction h a déclasser de l'information relativement à son premier arguement.*

Les fonctions de déclassement modélisent les possibillités offertes au programmeur pour faire des interférences entre haut et bas niveaux de confidentialité. On peut, d'un certain point de vue les voir comme des fonctions de cryptages dont le but est de rendre public une donnée après l'avoir encryptée.

Une des particularités de notre approche est qu'elle permet de définir des politiques de sécurités subtiles. On peut par exemple prendre en compte la

force du cryptage. Certains sont incassables (le cryptage vernam) alors que d'autres sont plutôt transparents (le codage de César) voir [Sin00]. Enfin, la force de certain dépend de la taille de la clef utilisée pour crypter. Si on prend l'exemple du code RSA [RSA78] alors une clef de 100 chiffres sera plus sûre qu'une clef de 50 chiffres. Il est possible d'encoder de telles subtilités dans les politiques de confidentialité. Considérons une fonction RSA d'arité 2 dont le premier argument est la clef et le second le message qui doit être encrypté. On peut considérer le treillis de confidentialité, construit sur les niveaux $\mathfrak{L} = \{\bot, \top, \top_1, K_{50}, K_{100}\}$, suivant :



L'idée derrière cette définition étant que K_{50}, K_{100} représentent les niveaux de confidentialité associés aux clefs d'encryptage, alors que $\top, \top_1$ et $\bot$ sont des niveaux de confidentialités associés aux données à encrypter. On peut alors définir la politique de sécurité suivante :

$$RSA(K_{100}, y) \to \bot$$
$$RSA(K_{50}, x) \to \top_1$$
$$RSA(\top_1, x) \to x$$
$$RSA(\top, x) \to x \quad RSA(\bot, x) \to x$$

Les trois dernière règles sont données à cause des conditions énoncées dans la définition 18. Ces règles ne sont pas supposées être utilisées en pratique dans cet exemple où l'on peut (par typage) requérir que le premier argument de la fonction RSA doit être un niveau de confidentialité étant soit K_{50} soit K_{100}.

Définition 19 (Niveau de confidentialité d'un terme) *Soit t un terme. Le niveau de confidentialité de t par rapport à une politique de confidentialité $\mathcal{SP}$, dénoté par $\pi(t)$ est la forme normale de $\sigma(t)!_{SP}$, avec σ qui est la substitution qui instancie chaque variable de t par $\bot$.*

Si on reprend la politique de confidentialité de l'exemple 1 le niveau de confidentialité de $f(g(h(x, y)), z)$ est la forme normale de $f(g(h(\bot, \bot)), \bot)$ qui est en l'occurrence $\top$. Notons que comme $\mathcal{SP}$ ne définit que des fonctions monotones (cf. définition 18), le niveau de confidentialité d'un terme est le niveau minimal qu'on peut obtenir quand on clot le terme.

La notion de niveau de confidentialité est étendue aux règles de réécritures et aux actions comme dans la définition 5, le niveau d'une règle est celui de sa partie gauche. On étend aussi la notion de niveau de confidentialité sur les équations de la forme $t_1 = t_2$ par :

$$\pi(t_1 = t_2 \wedge ... \wedge t_n = t_{n-1}) = \bigsqcap_{i=1}^{n} \pi(t_i)$$

Nous introduisons la notion de terme déclassé. Il s'agit d'un terme dont le niveau de confidentialtité est inférieur à celui de certains de ces sous-termes. Ce sont des termes qui, potentiellement, présentent des fuites d'information.

Définition 20 (Termes déclassés) *Soit $t = f(t_1, \ldots, t_n)$ avec $n \geq 1$ un terme. Ce terme est dit* déclassé *si :*

1. *Soit il existe $j \in \{1, \ldots, n\}$ tel que $\pi(t_j) \not\sqsubseteq \pi(t)$, ou bien*
2. *t est un sous-terme d'un terme déclassé.*

Dans le terme $t = h(f(k), g(f(k)))$, et étant donné la politique de confidentialité suivante :

$$\begin{array}{ll} k \rightarrow \top & f(x) \rightarrow \top \\ g(x) \rightarrow \bot & h(x, y) \rightarrow \top \end{array}$$

Le terme $t_{|1}$ n'est pas déclassé, $t_{|2}$, $t_{|2.1}$ et $t_{|2.1.1}$ sont déclassés. On peut noter que $t_{|1}$ et $t_{|2.1}$ représentent la même donnée mais n'ont pas le même statut concernant le déclassement.

Sûreté des processus vis-à-vis d'une politique de confidentialité

Nous commençons par la définition habituelle de règles sûres. Il ne doit pas y avoir de flot d'une partie gauche à la partie droite de la règle, donc le niveau de confidentialité ne peut que baisser lors d'un pas de réécriture. C'est à dire que le calcul de la forme normale d'un terme de confidentiatlité π n'utilise que des termes de niveau de confidentialité qui sont inférieurs ou égaux à π.

Définition 21 (Règles sûres) *Une règle de réécriture $l \rightarrow r \mid c$ est* sûre *quand les conditions suivantes sont satisfaites :*

1. *$\pi(\sigma(r)) \sqsubseteq \pi(\sigma(l))$ et $\pi(\sigma(c)) \sqsubseteq \pi(\sigma(l))$ pour toute substitution fermante $\sigma : \mathcal{V}ar(l) \rightarrow \mathfrak{L}$*
2. *l est déclassé quand r contient des sous-termes declassés.*

En effet les règles de la forme $SPY \to PIN$, étant donnée une politique de sécurité où $SPY \to \bot, PIN \to \top$, n'est pas une règle acceptable car elle rend publique directement la valeur d'une information privée. Cette remarque s'applique également de manière plus élaborée sur des règles de la forme suivante $SPY \to true \mid PIN = true$. La condition (2) n'est pas obligatoire. Elle assure que dans une dérivation de réécriture le descendant d'un terme ou sous-terme t est déclassé seulement si t lui même est déclassé.

Nous ne considérons que des stores ayant des règles sûres.

La notion d'équivalence entre les stores est la même que celle donnée dans la définition 7 mutatis mutandis (principalement la fonction de confidentialité ℓ est remplacée par la politique de sécurité $\mathcal{SP}$).

Nous introduisons une notion d'évaluation spécifique, l'évaluation de déclassement pour pouvoir tenir compte de la politique de sécurité : c'est cette évaluation qui sera la base de l'équivalence entre processus.

Définition 22 (Evaluation de déclassement) *Soient F, F' deux stores. $\mathfrak{s}\_eval(F, F', t)$ est la forme normale de t caculée en utilisant la réduction $\overset{F,F'}{\to}$ définie comme suit :*
$t \overset{F,F'}{\to}_{u,l\to r|c} t'$ telle que
- *$l \to r \mid c \in F'$ si la position de u est descendante d'un terme déclassé,*
- *$l \to r \mid c \in F$, sinon.*

En d'autres termes $\mathfrak{s}\_eval(F, F')$ est l'évaluation qui calcule les parties déclassées (et leur descendants) d'un terme en utilisant les règles de F', et les autres parties en utilisant les règles dans F. Si on suppose que les deux stores sont des SRT confluents, alors l'évaluation de déclassement est elle aussi confluente à cause de la condition sur la bonne formation des règles stipulant que les parties droites contiennent des termes déclassés si la partie gauche est un terme déclassé (cf. définition 21). Cela assure que chaque sous-terme déclassé est un descendant d'un sous-terme déclassé déjà présent dans le terme à normaliser.

Définition 23 (Sémantique opérationelle de déclassement) *Soient F et F' deux stores, et p terme de processus. La sémantique opérationnelle de déclassement est définie comme une relation de transition $\xrightarrow{F'}$. Les transitions sont de la forme $\langle F, p\rangle \xrightarrow{F'} \langle F'', p'\rangle$. Les règles sont données dans la figure 2.1, sauf pour les règles $(ea_{:=})$ et (P_{guard}) où $eval(F, t)$ est remplacé par $\mathfrak{s}\_eval(F, F', t)$.*

Définition 24 (Terme de processus sûr) *Soit p un terme de processus clos, $\pi \in \mathfrak{L}$, et deux stores π-équivalents F_1, F_2. p est dit sûr relativement à π et F_1, F_2 si $\langle F_1, p\rangle \longrightarrow \langle F_1', p'\rangle$ implique :*

- *Soit $\exists F_2'$ tel que $\langle F_2, p\rangle \xrightarrow{F_1} \langle F_2', p'\rangle$, $F_1' \equiv_\pi F_2'$ et p' sûr relativement à π et aux stores F_1', F_2'.*
- *ou alors $\langle F_2, p_2\rangle \not\xrightarrow{F_1}$ et pour tout $F_1^\sharp, p_1^\sharp$ tels que $\langle F_1', p_1'\rangle \longrightarrow^* \langle F_1^\sharp, p_1^\sharp\rangle$ on a, $F_1^\sharp \equiv_\pi F_2$.*

Donc un processus est sûr relativement à π si durant son exécution sur des données avec un niveau de confidentialité plus grand que π il n'y a pas d'influence sur les données de niveau inférieur à π sauf quand cette influence se fait par le truchement d'une fonction autorisée par la politique de confidentialité. Pour cela nous proposons d'évaluer les termes déclassifiés sur un store commun. Même si la définition n'est pas symétrique, la distinction apparente entre F_1 et F_2 n'est pas signifiante car la définition 25 doit être vérifiée pour n'importe quels stores, F_1, F_2, appropriés.

Définition 25 (Terme de processus fermé sûr) *Un terme de processus fermé* p *d'un système $\langle F, \Pi\mathcal{R}\rangle$ est dit* sûr *vis-à-vis d'une politique de sécurité $\mathcal{SP}$, ssi pour tout π, pour tous stores F_1, F_2 tels que $F \equiv_\pi F_1 \equiv_\pi F_2$,* p *est sûr relativement à π et aux stores F_1, F_2.*

Exemple 2 (Communication cryptée) *Supposons que nous ayons deux fonctions (d'arité un pour des raisons de simplicité) crypt et decrypt pour l'encodage/décodage, et considérons la politique de sécurité suivante :*
$crypt(x) \to \bot$ et $decrypt(x) \to \top$ qui signifie que les données encodées acquièrent un statut public (niveau de confidentialité $\bot$) et que les données décodées deviennent secrètes (niveau $\top$). Alors la communication d'un secret à travers un canal public entre deux processus peut s'écrire de la manière suivante :

$$\boxed{\begin{array}{ll} \alpha \;\Leftarrow & \left[c_\alpha = \mathbf{0} \Rightarrow \begin{array}{l} pub := crypt(PIN)\,; \\ done := \mathbf{0}\,;\; c_\alpha := \mathbf{1} \end{array}\right];\; \alpha' \\[2ex] \beta \;\Leftarrow & \left[done = \mathbf{0} \Rightarrow \begin{array}{l} k := decrypt(pub)\,; \\ done := \mathbf{1} \end{array}\right];\; \beta' \end{array}}$$

en étendant la politique de sécurité par : $pub \to \bot, c_\alpha \to \bot, done \to \bot$ et $PIN \to \top, k \to \top$. Alors en partant d'un store dans lequel done et c_α sont deux constantes définies par les règles $done \to \mathbf{1}, c_\alpha \to \mathbf{0}$, le terme de processus $\alpha \parallel \beta$ communique le secret PIN au moyen du canal public pub en utilisant la fonction d'encodage.

Le point important à relever dans cet exemple est le suivant. Imaginons qu'un processus tiers, disons un espion, γ tourne en parallèle avec α, β. Alors γ a accès à pub et peut l'utiliser pour modifier ses propres comportements de bas niveau de confidentialité. Il y a donc formellement une interférence entre une valeur de haut niveau, PIN, et un comportement de bas niveau. Par exemple si γ est défini par :

$$\boxed{\begin{array}{lll} \gamma & \Leftarrow & [pub = 1 \Rightarrow y := 1] \\ & & [pub \neq 1 \Rightarrow y := 2] \end{array}}$$

avec y tel que $y \rightarrow \bot$ soit dans la politique de sécurité, alors $\alpha \parallel \beta \parallel \gamma$ va présenter une fuite d'information. En effet en partant de deux stores $\bot$-équivalents la valeur de y peut finir par être différente. Il y a donc une différence entre les observations d'un niveau $\bot$ à partir du moment où $encrypt(pin)$ a une valeur différente dans les deux stores de départ.

Néanmoins, cette interférence est acceptable car les données échangées au moyen de pub sont encodées et cela est prévu dans la politique de sécurité. Il est aisé de vérifier que $\alpha \parallel \beta \parallel \gamma$ est un terme de processus clos sûr, notamment par le fait que $encrypt(pin)$ donnera la même valeur à cause de la définition de l'évaluation de déclassement $\mathfrak{s}\_eval(F, F', encrypt(pin))$.

Symétriquement on peut remarquer qu'à cause de la politique de confidentialité, une fois décodée, la donnée ne peut plus être utlisée de manière inapropriée, car le niveau de cette donnée (k dans notre exemple) redevient $\top$.

Analyse de sûreté vis-à-vis d'une politique de confidentialité

Pour analyser si un processus est sûr nous allons reprendre la technique développée dans un cadre multiparadigme pour la non-interférence (section 2.2.1) en l'adaptant pour tenir en compte la politique de confidentialité. Le cheminement est le même : il s'agit de définir une interprétation abstraite consistera à collecter des inéquations. La différence principale tient au fait qu'il n'est pas la peine de définir des termes abstraits auxquels on appliquerait une fonction de confidentialité. Il suffit de reprendre le terme lui même et de calculer sa valeur dans $\mathcal{SP}$ pour obtenir son niveau de confidentialité.

Nous définissons une sémantique opérationnelle abstraite pour un système C, et une politique de confidentialité $\mathcal{SP}$. L'abstraction d'un store est un ensemble d'inéquations sur des termes de $\mathcal{SP}$, sinon c'est mutatis mutandis la définition 11. De manière similaire à la définition 16 on définit une notion de compatibilité entre store et store abstrait.

La sémantique opérationnelle abstraite collecte des inéquations sur des termes de $\mathcal{SP}$ (cf. la définition de la sémantique opérationnelle abstraite en

section 12), tout comme dans la section 2.2.1.

Théorème 3 *Soit $C = \langle F, \Pi\!R \rangle$ un composant, p un terme de processus clos C, $\mathcal{SP}$ une politique de confidentialité sur F, et $\mathsf{p}^{\natural}_F$ le squelette de p. Si $\mathsf{p}^{\natural}_F$ est compatible avec $\mathcal{SP}$, alors p est un processus sûr vis-à-vis de $\mathcal{SP}$.*

2.3 Confidentialité et concurrence : approche par typage

2.3.1 Un λ-calcul avec resources adressées

$\lambda_{\mathfrak{ar}}$ est une version légèrement modifiée du calcul bleu de Boudol [Bou97].

La première modification que nous introduisons concerne les réductions : la récupération des données, c'est à dire la substitution d'une adresse par son contenu, peut se faire n'importe où dans le terme contrairement au calcul bleu où il ne peut avoir lieu que dans une variable de tête. Nous considérons plus ou moins la récupération des données comme une β-réduction où il n'y a pas de lieur et pour laquelle il y a un aspect concurrentiel concernant l'argument. La seconde modification concerne la gestion de l'opérateur parallèle : nous introduisons deux opérateurs différents. Le premier est restreint à l'expression du parallélisme entre processus alors que le second est utilisé pour représenter le parallélisme entre les données et les processus.

Le langage $\lambda_{\mathfrak{ar}}$ possède deux classes de termes : les termes et les ressources adressées.

t	::=	x, a	variables, addresses
	|	$(t\ \ t)$	application
	|	$\lambda x.t$	abstraction
	|	$t \parallel t$	parallélisme entre processus
	|	$\nu a(t)$	nouvelle addresse
	|	$t \mid \mathsf{s}$	processus et ressource adressée 1
	|	$\mathsf{s} \mid t$	processus et ressource addressée 2
s	::=	$\langle a \Leftarrow t \rangle$	ressource linéaire
	|	$\langle a = t \rangle$	ressource infinie
	|	$\mathsf{s} \mid \mathsf{s}$	parallélisme de ressource

Une ressource adressée est un terme défini à une certaine adresse. Dans l'expression $\langle a \Leftarrow t \rangle$, a est l'*addresse* et t la *ressource*. La ressource peut être linéaire ou infinie. Un ressource linéaire disparait quand elle est utilisée, alors

qu'une ressource infinie peut être utilisée un nombre de fois non borné. L'analogie est la même qu'entre l'implication linéaire et l'implication classique (cf. [Gir87]). Les ressources adressées peuvent être regroupées en utilisant l'opérateur |. Les addresses peuvent être vues comme des alias ou plus intuitivement comme des hyperliens et les ressources addressées comme des pages web.

Les seuls lieurs dans $\lambda_{\mathfrak{ar}}$ sont λ et ν. Dans $\langle a \Leftarrow t \rangle$, par exemple, le nom a n'est pas lié. Les *noms libres* sont définis comme il est de coutume en considérant ces lieurs.

Un contexte est un terme contenant une constante spéciale, $\Box$. On utilisera les lettres capitales $A, B, C, \ldots$ pour dénoter les contextes. Dans un contexte $C[\Box]$, $\Box$ peut être remplacé par un terme t ce qui donne le terme dénoté par $C[t]$ (on notera que les variables libres de t peuvent devenir liées dans $C[t]$).

On définit une équivalence structurelle $\equiv$ entre $\lambda_{\mathfrak{ar}}$-terms. Cette équivalence permet de considérer les termes comme une solution chimique à la [BB92] où processus et ressources adressées peuvent se mouvoir et interagir entres elles. Les règles d'équivalence structurelles sont données dans la figure 2.2.

Les règles structurelles de commutativité, d'associativité de migration de portée permettent de présenter les termes de $\lambda_{\mathfrak{ar}}$ de manière canonique.

On appelle *agents* les termes définis par :

$$Ag ::= a \mid x \mid \lambda x.t \mid (t \;\; t)$$

où t est un terme de $\lambda_{\mathfrak{ar}}$.

Par exemple le terme $(\lambda x.x \;\; (t \parallel u))$ est un agent, alors que le terme $t \parallel (f \;\; \lambda x.(x \;\; x))$ ne l'est pas. On peut remarquer que les agents ont la forme de λ-termes mais qu'ils n'en sont pas.

Fait 1 (Forme canonique) *Tout terme t de $\lambda_{\mathfrak{ar}}$ est structurellement équivalent à un terme en* forme canonique *:*

$$\begin{array}{ll} \nu a_1, \ldots, a_n & \\ \quad (t_1 \parallel \ldots \parallel t_p & \textit{agents} \\ \quad \mid \langle b_1 \Leftarrow v_1 \rangle \mid \ldots \mid \langle b_q \Leftarrow v_q \rangle & \textit{ressources linéaires} \\ \quad \mid \langle b_1 = v_1 \rangle \mid \ldots \mid \langle b_q = v_q \rangle) & \textit{ressources infinies} \end{array}$$

Il y a deux types de réductions $\lambda_{\mathfrak{ar}}$ dans : la β-réduction et la substitution d'une adresse par la ressource définie à cette adressse nommée *récupération des données.*

Définition 26 (Règles de réduction)

$$\begin{array}{lll}(\lambda x.t \;\; u) & \to_\beta & t\{x := u\} \\ t \mid \langle a \Leftarrow u \rangle & \to_\rho & t\{a := u\}\end{array}$$

pour $\to_\rho$*, il faut que* $a \in \mathsf{fn}(t)$ *et* t *soit un agent. On note* $\to$ *pour* $\to_\beta \cup \to_\rho$*.* $\to$ *est contextuellement fermée par une règle de contexte, et étendue à une congruence :*

$$\begin{array}{rcl} t \to t' & \Longrightarrow & C[t] \to C[t'] \\ t \to t' \text{ et } t \equiv u & \Longrightarrow & u \to t' \end{array}$$

Il est clair que ni la β-réduction ni la ρ-réduction normalisent. Le terme suivant :

$$\Omega = u \mid \langle u = u \rangle$$

se ρ-réduit indéfiniment sur lui même par exemple.

$\lambda_{\mathfrak{ar}}$ permet l'encodage du non déterminisme par :

$$c \mid \langle c \Leftarrow t \rangle \mid \langle c \Leftarrow u \rangle \tag{2.1}$$

avec $c \notin \mathsf{fn}(t) \cup \mathsf{fn}(u)$. Ce terme peut se réduire soit sur $t \mid \langle c \Leftarrow u \rangle$ ou bien sur $u \mid \langle c \Leftarrow t \rangle$. On peut facilement en tirer un encodage d'un opérateur de choix non-déterministe par :

$$\oplus \equiv \lambda x, y.\nu c(c \mid \langle c \Leftarrow x \rangle \mid \langle c \Leftarrow y \rangle)$$

qu'il est impossible de définir en λ-calcul.

On peut noter qu'une autre sorte de non-déterminisme est possible. Considérons le terme suivant :

$$t \equiv t_1 \parallel t_2 \parallel \ldots t_n \mid \langle a \Leftarrow v \rangle \tag{2.2}$$

où $a \in \mathsf{fn}(t_i)$ pour $i \in \{1, \ldots, n\}$. t'_i dénotant le terme :

$$t'_i \equiv t_1 \parallel t_2 \parallel \ \ldots \ \parallel t_i[a := v] \parallel \ \ldots \ \parallel t_n$$

pour tout $i \in \{1, \ldots, n\}$, $t \to t'_i$.

Ces deux formes de non-déterminismes, celle provenant de la commutativité de $\parallel$ et celle étant issue de la commutatitivité de $\mid$ sont différentes. L'une est basée sur la multiplicité des producteurs alors que l'autre l'est sur la multiplcté des consommateurs. Le non-déterminisme exhibé dans (2.2) rend bien compte de l'intuition qu'il y a derrière le parallélisme vu comme entrelacement. Si n processus sont éxécutés en parallèle, on ne peut pas savoir quel

sera le premier qui aura accès à la resource v située à l'adresse a. D'un autre côté, le non-déterminisme engendré par (2.1) est d'une autre nature : il s'agit de deux ressources ayant la même adresse, et c'est au processus de "choisir" quelle définition va être utilisée. Ce non-déterminisme n'est pas intrinsèque aux systèmes concurrents.

Dans $\lambda_{\mathfrak{ar}}$ on peut modéliser les communications dans un style à la π-calcul. Considérons le terme suivant :

$$\langle a \Leftarrow \lambda x.t \rangle \mid (a \;\; v) \rightarrow_{\rho} (\lambda x.t \;\; v) \rightarrow_{\beta} t\{x := v\}$$

Il permet de simuler une communication telle qu'elle a lieu en π-calcul :

$$a?(x).t \parallel a!\langle v \rangle \rightarrow t\{x := v\}$$

où $a?(x).t$ est le processus qui reçoit la valeur v sur le canal a et qui ensuite se comporte comme $t\{x := v\}$, et où $a!\langle v \rangle$ est le processus émettant la valeur v sur le canal a. Néanmoins, dans le contexte du π-calcul seuls des noms de canaux peuvent être échangés, alors que dans $\lambda_{\mathfrak{ar}}$ il est possible d'échanger n'importe quels termes.

Ces quelques exemples, bien que très simples, permettent de mettre en lumière d'intéressantes caractéristiques de $\lambda_{\mathfrak{ar}}$. La première est une justification des deux opérateurs parallèles différents. Nous avons vu que cela conduit de manière directe à deux formes de non-déterminisme. Cela suggère qu'il y a plus qu'une simple distinction syntaxique entre les deux opérateurs. La seconde est que $\lambda_{\mathfrak{ar}}$ est un calcul plus fin que le π-calcul dans le sens qu'une réduction atomique du π-calcul est interprétée en $\lambda_{\mathfrak{ar}}$ par une ρ-réduction suivit d'une β-réduction. la ρ-réduction modélisant le chargement d'une information distante alors que la β-réduction prend en charge le passage de paramètre (subsitution d'un paramètre par le paramètre réel). Ces deux étapes sont confondues en π-calcul alors qu'elles sont clairement identifiées en $\lambda_{\mathfrak{ar}}$.

2.3.2 Typage de $\lambda_{\mathfrak{ar}}$

$\lambda_{\mathfrak{ar}}$ est un calcul très riche. Nous proposons une discipline de typage qui limite le nombre de termes bien formés. Les types seront utilisés comme des squelettes qu'on pourra utiliser, en les annotant, pour faire des analyses statiques.

Les types de $\lambda_{\mathfrak{ar}}$ sont les types du λ-calcul (types de base et types flèches), avec l'addition d'un type parallèle : $\mathsf{Pa}(\sigma, \tau)$.

Définition 27 (Types) *Les types sont définis par :*

$$\begin{array}{rcll} \tau & ::= & b & \textit{Type de base} \\ & | & \circ & \textit{Type enregistrement} \\ & | & \tau \to \tau & \textit{Type flèche} \\ & | & \mathsf{Pa}(\tau,\tau) & \textit{Type parallèle} \end{array}$$

A cause de l'équivalence structurelle entre les termes, il faut définir une relation d'équivalence sur les types. On définit la relation $\equiv_{\mathbb{T}}$ entre types comme étant la plus petite relation d'équivalence contenant la relation R définie par :

Soient τ, σ, γ dans $\mathbb{T}$, alors $R(\mathsf{Pa}(\tau,\sigma), \mathsf{Pa}(\sigma,\tau))$ et,
$R(\mathsf{Pa}(\tau, \mathsf{Pa}(\sigma,\gamma)), \mathsf{Pa}(\mathsf{Pa}(\tau,\sigma),\gamma))$.

Dans le reste nous travaillons modulo $\equiv_{\mathbb{T}}$ équivalence. C'est à dire que quand nous écrivons $\tau = \sigma$ nous entendons en fait $\tau \equiv_{\mathbb{T}} \sigma$.

Nous étendons la syntaxe de $\lambda_{\mathfrak{ar}}$ afin de mettre des annotations de type pour les variables liées. Nous développons un système de typage à la Church.

$$t ::= \ldots \mid \nu a{:}\tau(t) \mid \lambda x{:}\tau.t \mid \ldots$$

le reste des termes est inchangé.
Les contextes de typages sont des listes de paires assignant un type à une variable ou une adresse :

$$\Gamma ::= \langle\rangle \mid \Gamma, x : \tau$$

où $\langle\rangle$ est la liste vide.

Les jugements de typage sont de la forme $\Gamma \vdash t : \tau$. Les règles de déductions sont données dans la figure 2.3.

Une particularité de ce système de typage est que les types y suivent la forme des termes, tout comme en programmation fonctionnelle, ce qui n'est par contre pas habituel pour ce genre de calculs. Dans les algèbres de processus les règles de typages sont plutôt de la sorte :

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \tau}{\Gamma \vdash t \mid u : \tau}$$

il y est donc difficile de se servir des types pour analyser les interactions entre t et u : car ils doivent être de même type.

Il n'est pas possible de définir des types équivalents à ceux de $\lambda_{\mathfrak{ar}}$ en π-calcul car le nombre de processus en parallèle peut changer suite à, n'importe quel pas de communication : la forme des termes évolue pendant le calcul. Si l'on combine cela au non-déterminisme des algèbres de processus il devient impossible d'assurer une quelconque stabilité du typage lors de la réduction. Cela est possible dans $\lambda_{\mathfrak{ar}}$ notamment grâce aux clarifications faites par rapport au π-calculs dans lequel différents concepts sont confondus.

Les propriété standard du typage (au sens typage des langages fonctionnels) sont valides dans $\lambda_{\mathfrak{ar}}$.

Lemme 3 (Unicité du type) *Si* $\Gamma \vdash t : \tau$ *et* $\Gamma \vdash t : \tau'$ *alors* $\tau = \tau'$.

Théorème 4 (Equivalence structurelle) $\Gamma \vdash t : \tau$ *et* $t \equiv u$ *implique* $\Gamma \vdash u : \tau$.

Lemme 4 (Affaiblissement du contexte) $\Gamma \vdash t : \tau$ *et* $x \notin \Gamma$ *implique* $\Gamma, x : \sigma \vdash t : \tau$.

Lemme 5 (Lemme de substitution) $\Gamma \vdash x : \tau$, $\Gamma \vdash u : \tau$ *et* $\Gamma \vdash t : \sigma$ *implique* $\Gamma \vdash t\{x := u\} : \sigma$.

Théorème 5 (Stabilité du typage par réduction) $\Gamma \vdash t : \tau$ *et* $t \to t'$ *implique* $\Gamma \vdash t' : \tau$.

On peut remarquer que que le typage n'assure pas la normalisation forte. En effet, il est possible de typer le terme Ω défini ci dessus. Pour tout type σ, $u : \sigma \vdash \langle u = u \rangle : \circ$, et $u : \sigma \vdash u : \sigma$, d'où par la règle $PD2$ nous pouvons conclure $u : \sigma \vdash \Omega : \sigma$.

Considérons le terme $t = u \mid \langle u = u \parallel u \rangle$. Ce terme se réduit indéfiniment par :

$$\begin{aligned} t &\to u \parallel u \mid \langle u = u \parallel u \rangle \\ &\to u \parallel u \parallel u \mid \langle u = u \parallel u \rangle \\ &\to \dots \end{aligned}$$

Le terme t ne peut pas être bien typé. En effet, si τ est le type de u. Alors $u \parallel u$ est de type $\mathsf{Pa}(\tau, \tau)$ (à cause de la règle $PPAR$) qui est différent de τ. Donc $\langle u = u \parallel u \rangle$ ne peut pas être correctement typé : la règle $STO2$ oblige l'adresse et le terme qui est situé à cette adresse à avoir le même type. Cet exemple illustre bien pourquoi le type reste stable par réduction.

2.3.3 Abstraction de sorte et analyse de non-interférence

Dans [Pro00] nous avons présenté un mécanisme générique pour modifier les systèmes de typages fonctionnels pour faire des analyses de non-interférence basées sur le typage. L'idée est que l'analyse de programme basée sur les types revient à abstraire vis à vis de sortes (intuitivement le type des types). Par exemple le jugement $\mathsf{Int} : *$ signifie que le type des entiers Int est de sorte $*$. Dans [Pro00] nous n'avons considéré que deux sortes différentes

$\top$ et $\bot$. Cela conduit à deux types entiers différents : $\mathsf{Int}^{\top} : \top$ et $\mathsf{Int}^{\bot} : \bot$. De plus nous introduisons une notion de variable de sorte, $\mathsf{Int} : \alpha$ signifiant qu'il s'agit du type des entiers sur la sorte *variable* α qui peut être soit $\top$ soit $\bot$. On peut alors définir l'abstraction et l'application relativement à ce type de variables. Il est possible de prouver que les parties typées avec la sorte $\top$ et les parties typées sur la sorte $\bot$ n'interfèrent pas.

Nous allons suivre cette idée et définir un nouveau langage, $\lambda_{\mathfrak{ar}}\mathcal{E}$, à partir de $\lambda_{\mathfrak{ar}}$, pour lequel il est possible d'abstraire par rapport aux sortes. Il s'agit donc d'une application plus ou moins directe de [Pro00] mais dans un cadre concurrent ce qui change un peu la technique de preuve eut égard à la définition du langage et de l'équivalence observationnelle.

Les sortes, le type des types, sont définies par :

$$\mathfrak{s} ::= \top \mid \bot \mid \alpha$$

où α représente les variables de sortes.

Nous modifions la version typée de $\lambda_{\mathfrak{ar}}$: les sortes peuvent apparaître dans les termes typés. Les types étendus sont définis par :

$$\tau := b^{\mathfrak{s}} \mid \circ \mid \tau \to \tau \mid \mathsf{Pa}(\tau, \tau) \mid \forall \alpha.\tau$$

En outre de la définition de types étendus (polymorphismes sur les sortes), nous avons ajouté des annotations sur les types de base. Ces annotations dénotent la sorte du type de base. la manière la plus intuitive de le voir est d'imaginer qu'il y a deux classes de données : privées et publiques (comme dans [Aba97]).

De concert avec les types étendus nous introduisons les contextes étendus qui sont des listes de paires qui sont soit des variables de termes ou des variables de sortes avec un type ou une sorte.

$$\Gamma \vdash \langle\rangle \mid \Gamma, x : \tau \mid \Gamma, \alpha \mid \Gamma$$

Il est à noter que l'ordre des déclarations est maintenant signifiant. En effet les types peuvent dépendre de variables de sortes, par exemple : $\Gamma = \alpha, x : \mathsf{Int}^{\alpha}$. Pour contrôler la formation des contextes nous introduisons un jugement de bonne formation des contextes, $\Gamma \vdash$, et de bonnes formations des types $\Gamma \vdash \tau$ et des sortes $\Gamma \vdash \mathfrak{s}$. Les règles de ces jugements, ainsi que les règles de typages supplémentaires pour les termes sont données dans la figure 2.4.

Les méta-propriétés examinées dans 2.3.1 pour $\lambda_{\mathfrak{ar}}$ restent vérifiées dans $\lambda_{\mathfrak{ar}}\mathcal{E}$.

D'un point de vue intuitif, l'introduction des sortes $\bot$ et $\top$ produit deux variantes de $\lambda_{\mathfrak{ar}}$. Dans une version tout est annoté avec $\top$, et dans l'autre

version par $\bot$. En réalité les deux variantes coexistent car il est possible de construire des termes ayant des sous-termes de type ayant pour sorte $\bot$ et d'autres $\top$. Par exemple considérons le contexte $f : \mathsf{Int}^{\bot} \to \mathsf{Int}^{\top}, x : \mathsf{Int}^{\bot}$, alors on peut construire le terme :

$$(f \;\; x) \mid \langle x \Leftarrow 5^{\bot} \rangle \mid \langle f \Leftarrow \lambda y{:}\mathsf{Int}^{\bot}.4^{\top} \rangle$$

Les types, les termes et ressources $\Gamma/\mathfrak{s}$-saturés sont ceux dans lesquels les seules sortes apparaissant sont des variables liées ou bien la sorte $\mathfrak{s}$:

Définition 28 (Types, termes et adresses saturés) *Le type τ est $\Gamma/\mathfrak{s}$-saturé si :*

- $\tau = b^{\mathfrak{s}}$;
- *ou* $\tau = \tau_1 \to \tau_2$ *ou* $\mathsf{Pa}(\tau_1, \tau_2)$ *et* τ_1, τ_2 *sont $\Gamma/\mathfrak{s}$-saturés ;*
- *ou* $\tau = \forall \alpha.\tau_1$ *et* $\tau_1\{\alpha := \mathfrak{s}\}$ *es $\Gamma/\mathfrak{s}$-saturé.*

Le terme t est $\Gamma/\mathfrak{s}$-saturé si :

- $\Gamma \vdash t : b^{\mathfrak{s}}$;
- *ou* $t = x$ *et* $\Gamma \vdash x : \tau$ *et* τ *est $\Gamma/\mathfrak{s}$-saturé ;*
- *ou* $t = (t_1 \;\; t_2)$ *ou* $t_1 \parallel t_2$ *et* t_1, t_2 *sont $\Gamma/\mathfrak{s}$-saturés ;*
- *ou* $t = \lambda x{:}\tau.t_1$ *ou bien* $\nu x{:}\tau(t_1)$ *et* τ *est $\Gamma/\mathfrak{s}$-saturé et* t_1 *est* $\Gamma, x : \tau/\mathfrak{s}$*-saturé ;*
- *ou* $t = t' \mid \mathsf{s}$ *ou* $t = \mathsf{s} \mid t'$*, et* t' *est $\Gamma/\mathfrak{s}$-saturé* s *est $\Gamma/\mathfrak{s}$-saturée.*

Une resource adressée s *est $\Gamma/\mathfrak{s}$-saturée si* $\mathsf{s} = \langle a = t \rangle$ *ou* $\mathsf{s} = \langle a \Leftarrow t \rangle$ *et t est $\Gamma/\mathfrak{s}$-saturé.*

Quand le contexte peut être inféré ou bien quand il n'a pas d'importance on écrit plus simplement $\mathfrak{s}$-saturé.

Un *test*, noté $\mathfrak{T}$, est un processus de type $\Theta^{\mathfrak{s}}$, le type des tests sur la sorte $\mathfrak{s}$, $\mathfrak{s}$ étant soit $\top$ soit $\bot$. Il n'y a qu'un seul habitant pour ce type : $\theta^{\mathfrak{s}}$, le succès. Nous avons les règles de typage suivantes :

$$\frac{\Gamma \vdash}{\Gamma \vdash \theta^{\mathfrak{s}} : \Theta^{\mathfrak{s}}}$$

Un test $\mathfrak{T}$ placé en parallèle avec un terme t interagit avec t, et $\mathfrak{T}$ peut dériver en θ comme résultat de l'observation. On dit qu'un terme t passe avec succès le $\mathfrak{s}$-test $\mathfrak{T}$ s'il existe une réduction $\mathfrak{T} \parallel t \to^* t'$ telle que t' est de la forme $\theta^{\mathfrak{s}} \parallel u$. Si t passe le $\mathfrak{s}$-test $\mathfrak{T}^{\mathfrak{s}}$, on écrit $t \Downarrow_{\mathfrak{s}}^{\mathfrak{T}}$.

Définition 29 (Equivalence observationnelle) *Soit $\Gamma \vdash t, u : \tau$, avec τ $\Gamma/\mathfrak{s}$-saturé, on a $t \cong_{\mathfrak{s}} u$ si pour tout test $\mathfrak{T}^{\mathfrak{s}}$ tel que $\Gamma \vdash \mathfrak{T} : \Theta^{\mathfrak{s}}$:*

$$t \Downarrow_{\mathfrak{s}}^{\mathfrak{T}} \iff u \Downarrow_{\mathfrak{s}}^{\mathfrak{T}}$$

Théorème 6 (Non-interférence) *Si* $\Gamma, x : \sigma \vdash t : \tau$*,* σ *est* $\bot$*-saturé et* τ *est* $\top$*-saturé, et* $\Gamma \vdash u, u' : \sigma$*, alors* $t[x := u] \cong_\top t[x := u']$*.*

Entre autre, ce théorème nous assure que des termes $\bot$-saturés ne peuvent pas interférer avec des termes $\top$-saturés. L'application à la confidentialité pouvant se faire de la façon suivante : quand un utilisateur reçoit un programme à éxécuter, il peut statiquement vérifier par typage que ce programme ne va pas accéder ses données privées (il suffit de saturer les termes publics et privés par des sortes différentes).

On peut remarquer que cette extension aurait pu aussi être réalisée en utilisant le système de typage originel du calcul bleu (voir [Bou97]). Cependant il y a une règle de typage de ce système :

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \tau}{\Gamma \vdash t \parallel u : \tau}$$

qui rend impossible la description des interférences entre t et u en utilisant une technique basée sur les sortes. D'un autre côté avec le système que nous avons présenté de telles interférences sont analysables. Par exemple soit $\tau = \mathsf{Int}^\top \to \mathsf{Int}^\bot \to \mathsf{Int}^\top$ et $\sigma = \mathsf{Int}^\top \to \mathsf{Int}^\top \to \mathsf{Int}^\top$. Alors si $\Gamma \vdash t : \tau$ et $\Gamma \vdash u : \sigma$, l'utilisation de la règle $PPAR$ permet de conclure que $\Gamma \vdash t \parallel u : \mathsf{Pa}(\tau, \sigma)$. Maintenant juste en regardant la forme des types, il est possible d'inférer qu'il n'y a pas d'interférences entre le second argument de u (qui est un entier sur la sorte $\bot$) avec le premier et le second argument de t'. Sur cet exemple simpliste il est possible de prouver que si une donnée privée est donnée comme second argument à u, alors t ne peut rien apprendre sur cette valeur. Par contre, il est possible qu'il y ait des interférences entre le premier argument de u et le résultat de t car ils sont typés de même sorte.

$$\langle F, \mathsf{tell}(R); a\rangle \hookrightarrow \langle F \cup R, a\rangle \quad (ea_{\mathsf{tell}})$$
$$\langle F, \mathsf{del}(R); a\rangle \hookrightarrow \langle F \setminus R, a\rangle \quad (ea_{\mathsf{del}})$$

$$\langle F, f := t; a\rangle \hookrightarrow \langle F \bullet (f \to eval(F,t)), a\rangle \quad (ea_{:=})$$
$$\langle F, \mathsf{skip}; a\rangle \hookrightarrow \langle F, a\rangle \quad (ea_{\mathsf{skip};})$$

$$\theta; p \equiv_p p \quad (Eq_{\theta;})$$
$$\theta \parallel p \equiv_p p \quad (Eq_{\theta\parallel})$$

$$p_1 + p_2 \equiv_p p_2 + p_1 \quad (Eq_{+\ com})$$
$$p_1 \parallel p_2 \equiv_p p_2 \parallel p_1 \quad (Eq_{\parallel\ com})$$

$$\frac{p_1 \equiv_p p_2 \quad \langle F, p_2\rangle \longrightarrow \langle F', p_3\rangle \quad p_3 \equiv_p p_4}{\langle F, p_1\rangle \longrightarrow \langle F', p_4\rangle} \quad (P_{\equiv_p})$$

$$\frac{\langle F, a_1; \ldots; a_n; \mathsf{skip}\rangle \hookrightarrow^* \langle F', \mathsf{skip}\rangle \quad eval(F,g) = \mathsf{True}}{\langle F, [g \Rightarrow a_1; \ldots; a_n]\rangle \longrightarrow \langle F', \theta\rangle} \quad (P_{guard})$$

$$\frac{(\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \Sigma_{j=1}^m \alpha_j \,; p_j) \in \Pi\mathbb{R} \qquad \langle F, (\Sigma_{j=1}^m \alpha_j \,; p_j)[x_i/t_i]\rangle \longrightarrow \langle F', p'\rangle}{\langle F, \mathrm{q}(t_1, \ldots, t_n)\rangle \longrightarrow \langle F', p'\rangle} \quad (P_{abs})$$

$$\frac{\langle F, p_1\rangle \longrightarrow \langle F', p_1'\rangle}{\langle F, p_1 + p_2\rangle \longrightarrow \langle F', p_1'\rangle} \quad (P_+) \qquad \frac{\langle F, p_1\rangle \longrightarrow \langle F', p_1'\rangle}{\langle F, p_1 \ \mathsf{op}\ p_2\rangle \longrightarrow \langle F', p_1' \ \mathsf{op}\ p_2\rangle} \ \mathsf{op} \in \{\parallel, ;\} \quad (P_{\mathsf{op}})$$

FIGURE 2.1 – Règles d'inférences de la sémantique opérationnelle

$t \parallel u \equiv u \parallel t$	$\mathsf{s}_1 \mid \mathsf{s}_2 \equiv \mathsf{s}_2 \mid \mathsf{s}_1$	commutativité
$t \mid \mathsf{s} \equiv \mathsf{s} \mid t$		
$(t \parallel u) \parallel r \equiv t \parallel (u \parallel r)$	$(t \parallel u) \mid \mathsf{s} \equiv t \parallel (u \mid \mathsf{s})$	associativité
$(\mathsf{s}_1 \mid \mathsf{s}_2) \mid t \equiv \mathsf{s}_1 \mid (\mathsf{s}_2 \mid t)$		
$\langle a = t \rangle \equiv \langle a = t \rangle \mid \langle a \Leftarrow t \rangle$		duplication
$\nu a(t) \parallel u \equiv \nu a(t \parallel u)$	$\nu a(t) \mid \mathsf{s} \equiv \nu a(t \mid \mathsf{s})$	migration de portée
$(a \notin \mathsf{fn}(u) \cup \mathsf{fn}(\mathsf{s}))$		
$t \equiv u \implies C[t] \equiv C[u]$		

FIGURE 2.2 – Equivalence structurelle

$$[HYP]\ \frac{}{\Gamma \vdash x : \tau}(x : \tau \in \Gamma) \qquad [NEW]\ \frac{\Gamma, a : \sigma \vdash t : \tau}{\Gamma \vdash \nu a{:}\sigma(t) : \tau}$$

$$[APP]\ \frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash (t\ \ u) : \tau} \qquad [ABS]\ \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x{:}\sigma.t : \sigma \to \tau}$$

$$[STO1]\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \langle a \Leftarrow t \rangle : \circ}(a : \tau \in \Gamma) \qquad [STO2]\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \langle a = t \rangle : \circ}(a : \tau \in \Gamma)$$

$$[PPAR]\ \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t \parallel u : \mathsf{Pa}(\tau, \sigma)}(\tau, \sigma \neq \circ)$$

$$[PD1]\ \frac{\Gamma \vdash \mathsf{s} : \circ \quad \Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{s} \mid t : \tau}$$

$$[PD2]\ \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \mathsf{s} : \circ}{\Gamma \vdash t \mid \mathsf{s} : \tau}$$

$$[DPAR]\ \frac{\Gamma \vdash \mathsf{s}_1 : \circ \quad \Gamma \vdash \mathsf{s}_2 : \circ}{\Gamma \vdash \mathsf{s}_1 \mid \mathsf{s}_2 : \circ}$$

FIGURE 2.3 – Règles de typage de $\lambda_{\mathfrak{ar}}$

– Contextes : $[CtxTyp]\ \frac{\Gamma \vdash \quad \Gamma \vdash \tau}{\Gamma, x : \tau \vdash}(x \notin \Gamma) \quad [CtxSor]\ \frac{\Gamma \vdash}{\Gamma, \alpha \vdash}(\alpha \notin \Gamma)$

– Sortes : $[Sor\top]\ \frac{\Gamma \vdash}{\Gamma \vdash \top} \quad [Sor\bot]\ \frac{\Gamma \vdash}{\Gamma \vdash \bot} \quad [SorVar]\ \frac{\Gamma \vdash}{\Gamma \vdash \alpha}(\alpha \in \Gamma)$

– Types : $[TpBase]\ \frac{\Gamma \vdash \mathfrak{s}}{\Gamma \vdash b^{\mathfrak{s}}} \quad [Tp \to]\ \frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \to \sigma} \quad [TpPar]\ \frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \mathsf{Pa}(\tau, \sigma)}$

$$[Tp\forall]\ \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\tau}$$

– Termes : $[ABSSor]\ \frac{\Gamma, \alpha \vdash t : \tau}{\Gamma \vdash (\alpha^{\mathcal{A}} t) : (\forall \alpha.\tau)} \quad [APPSor]\ \frac{\Gamma \vdash t : \forall \alpha.\tau \quad \Gamma \vdash \mathfrak{s}}{\Gamma \vdash (t \quad \mathfrak{s}) : (\tau\{\alpha := \mathfrak{s}\})}$

FIGURE 2.4 – Règles de typage pour $\lambda_{\mathfrak{ar}}\mathcal{E}$

Chapitre 3

Enchevêtrement et séparabilité

3.1 Physique quantique et programmation

L'idée d'utiliser les spécificités de la mécanique quantique pour calculer de manière performante repose sur une remarque de Feynmann dans [Fey82] : dans cet article il montre qu'il est impossible de simuler efficacement un système quantique sans faire face à une explosion combinatoire. Il propose alors de retourner la problématique et d'utiliser la puissance des phénomènes quantiques pour calculer rapidement. Les résultats de factorisation en temps polynomial de Shor [Sho94] et de recherche en racine carrée du nombre d'éléments dans une liste non ordonnée de Grover [Gro96] ont fondé la pertinence de cette remarque en offrant des accélérations respectivement exponentielles et quadratiques par rapport aux meilleurs algorithmes classiques connus. Dans le cas du résultat de Grover, l'accélération est impossible à obtenir de manière classique ; en ce qui concerne le résultat sur la factorisation, on ne connait pas actuellement la borne inférieure de la complexité de ce problème et s'il est possible ou non d'obtenir un algorithme classique de factorisation en temps polynomial.

Cette accélération est rendue possible par la conjonction de plusieurs particularités liées aux propriétés de la mécanique quantique. Le premier point est la possibilité de représenter l'information sous forme de bits quantiques qui ont la caractéristique de pouvoir être représentés comme une superposition de **0** et de **1** plutôt que d'être soit **0** soit **1**. La forme générale d'un bit quantique est : $\alpha \mid \mathbf{0}\rangle + \beta \mid \mathbf{1}\rangle$, où α, β sont des nombres complexes. L'accès à l'information contenu dans un bit quantique se fait au moyen d'une mesure dont le résultat est soit **0** soit **1**, les résultats étant obtenus avec une probabilité respective $|\alpha|^2$ et $|\beta|^2$. Une fois mesuré le bit quantique devient classique. Le second point s'appelle l'enchevêtrement, c'est un phénomène

mis en lumière par le paradoxe Einstein-Podolsky-Rosen [EPR35]. Il s'agit du fait qu'en agissant sur une particule d'un système cela peut avoir des conséquences sur celles qui sont "enchevêtrées" avec cette dernière. La définition mathématique de l'enchevêtrement est la suivante : l'espace de Hilbert des états d'un système à n bits quantiques est le produit tensoriel des espaces de Hilbert de chaque particule, or il existe des états qui ne peuvent être représentés comme un produit tensoriel de sous-espaces de Hilbert (voir [NC00]).

Pour simplifier, en combinant les deux aspects (superposition quantique et enchevêtrement) on arrive à mener des calculs en parallèles (faire les calculs sur des superpositions revenant en gros à faire les calculs sur plusieurs entrées possibles en une seule passe). Ces deux mécanismes forment en quelques sortes sorte le moteur des modèles de calcul quantique. Il y a bien d'autres particularités : axiome de non clonage (il est impossible, en toute généralité, de recopier un bit quantique), réversibilité des transformations, unitarité des opérations etc.

Notre but n'est pas d'étudier l'algorithmique et les propriétés spécifiques des calculs quantiques. Nous référons le lecteur à [NC00] concernant les spécificités des modèles de calcul basés sur la mécanique quantique. Notre problèmatique est de travailler sur les abstractions rendant la programmation quantique abordable pour un non-spécialiste de la physique quantique. En effet les phénomènes de superposition et d'enchevêtrement sont tout sauf intuitifs. D'ailleurs quand on observe de près comment les algorithmes quantiques sont définis il s'agit plus de "trucs et astuces" assemblés pour aboutir à un certain but qu'à l'application de principes généraux et abstraits tels qu'on les connait en informatique classique.

Notre objectif est donc d'isoler des concepts de haut niveau, d'identifier des paradigmes ou des abstractions utiles pour les programmeurs quantiques. Cette problématique de recherche est neuve mais a déjà reçu une attention marquée [Sel06].

Une proposition de langage, QML [AG05], a été faite pour représenter des structures de contrôles intégrant la superposition. Nous allons nous focaliser sur la seconde particularité quantique : la gestion et les implications de l'enchevêtrement. Il est prouvé que sans enchevêtrement, les calculs quantiques sont efficacement simulables de manière classique [Vid03]. Il s'agit donc d'une ressource cruciale mais difficile à comprendre d'un point de vue intuitif, comme en témoigne l'article d'Einstein Podolsky et Rosen qui visiblement n'arrivaient pas à y croire tellement cette propriété physique est contre-intuitive !

L'enchevêtrement des bits quantiques présente des similarités avec les problèmes que l'on rencontre lors de la manipulation des variables qui peuvent

être aliasées (via des manipulations de pointeurs). Très grossièrement les deux situations sont équivalentes car en agissant sur un bit quantique (resp. sur une variable de nom x) on peut agir sur un autre bit quantique (resp. sur une variable de nom y) si les bits quantiques sont enchevêtrés (resp. si x et y sont aliasées). D'un point de vue de programmeur on est confronté à un problème de non-compositionnalité : la sémantique d'un programme ne dépent pas uniquement du programme en tant que tel mais dépent aussi de son contexte (quels sont les bits quantiques enchevêtrés entre eux, quels sont les noms aliasés).

La notion de dépendance dans un calcul quantique est donc différente de celle dans un calcul classique du fait de cette enchevêtrement. Nous commençons en section 3.2 par nous intéresser aux problèmes engendrés par ces dépendances et ses implications en termes de portée des variables. Nous montrons comment l'utilisation de nouveaux lieurs, aux propriétés différentes de celles de la λ-abstraction, permettent de mieux gérer ce problème de portée (ces nouveaux lieurs rendent aussi possible l'écriture d'algorithmes quantique "abstraits" copiables alors que l'axiome de non clonage interdit normalement ce genre de pratique). Dans la section 3.3, nous développons une logique à la Hoare dans laquelle les assertions permettent de décrire quels sont les bit quantiques enchevêtrés ou, duallement séparables.

3.2 Compositionalité et lieurs

3.2.1 Un calcul d'état global

λ_{GS} : un calcul pour la gestion d'un état global

λ_{GS} est un calcul qui est défini relativement à un instrument physique dont on se sert pour faire des calculs en agissant dessus. D'un point de vue général cet instrument physique peut être de n'importe quel type : un mécanisme quantique, une solution ADN ou n'importe quel mécanisme de calcul "naturel". Les termes de λ_{GS} servent à contrôler l'artefact physique. Donc les programmes sont composés de deux parties : une partie "physique" (l'état global qui est une mathématisation de l'artefact utilisé pour le calcul) et une partie purement logicielle (le terme fonctionnel qui contrôle le processus physique de calcul).

La partie fonctionnelle de λ_{GS} est implantée par un λ-calcul standard étendu pour pouvoir gérer un état global. Les interactions avec l'état global sont gérées par deux mécanismes :

1. Adresses : du point de vue des termes les adresses sont justes des noms.

Du point de vue de l'état global les adresses sont les adresses des entités physiques (l'adresse concrète d'une information enregistrée dans une mémoire RAM, un bit quantique spécifique etc.).

2. Actions : du point de vue des termes les actions sont des termes qui peuvent être évalués mais dont l'évaluation dépent à la fois de l'état global et de la valeur des paramètres de l'action (par exemple le résultat de la mesure d'un bit quantique). Du point de vue de l'état global les actions sont des modifications concrètes de cet état (la mise à jour d'une cellule mémoire dans une RAM, l'application physique d'une porte unitaire à un bit quantique).

Définition de λ_{GS}

Un calcul λ_{GS} est défini relativement au choix de l'état global et des actions qui sont possibles.

Définition 30 (Termes) *L'ensemble des termes $\mathcal{T}$ est inductivement défini par :*

$$\begin{array}{rcl} M, N, P & ::= & x \mid \ell \mid \lambda x.M \mid (M \quad N) \\ & & \mid \langle M, N\rangle \mid \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N \\ & & \mid \nu x.M \mid \rho\ell.M \mid !M \\ & & \mid \mathsf{a}_i(M) \end{array}$$

où x, ℓ sont respectivement des noms de variable et d'adresse et sont des éléments d'un ensemble dénombrable $\mathcal{V}$. $\mathcal{L}$ est le sous-ensemble de $\mathcal{V}$ dénotant les adresses.

Il n'y a pas de distinction formelle entre x and ℓ. Cependant nous utiliserons la convention suivante : ℓ sera utilisé pour dénoter une adresse (donc une référence à l'état global) alors que x dénotera une λ-variable standard.

Les deux premières lignes de la définition 30 definissent un λ calcul pur avec des couples de termes ($\overrightarrow{M}$ dénote un n-uplet de termes : $\overrightarrow{M} \stackrel{def}{=} \langle M_1, \langle M_2, \langle \ldots, M_n\rangle \ldots\rangle\rangle$). Nous avons choisi d'avoir des n-uplets natifs pour λ_{GS}, car les actions ne peuvent pas être curryfiées : ce ne sont pas des fonctions au sens du λ-calcul.

Dans $\nu x.M$, x est une nouvelle adresse abstraite et est liée dans M. L'idée est que $\nu x.M$ est un morceau de code qui va utiliser l'état global pour faire un calcul et que l'adresse sera pointée par x.

Par contre $\rho\ell.M$ dénote un morceau de code qui utilise *concrètement* une adresse de l'état global. Dans $\rho\ell.M$ ℓ est lié dans M.

$!M$ est la *concrétisation* d'une action. Cela permet de transformer une adresse abstraite en une adresse concrète. Intuitivement (la définition formelle est donnée dans la définition 34) nous avons la réduction :

$$C[!\nu x.M] \to C[\rho\ell.M[x := \ell]]$$

où ℓ est une nouvelle adresse relativement à $FV(C[!\nu x.M])$. L'idée étant que la concrétisation a un effet de bord sur l'état global : cela crée concrètement une nouvelle ressource.

a_i sont les actions possibles, cet ensemble d'action est relatif à l'état global. Par example, si l'état global est une mémoire enregistrant des termes alors les actions usuelles sont l'affectation et la déréférence de pointeur. Si l'état global est composé de bits quantiques, alors les actions typiques comprennent les portes unitaires (par exemple la phase, la négation conditionnelle et la transformation d'Hadamard ainsi que la mesure).

Dans λ_{GS}, il y a trois lieurs λ, ν et ρ.

Définition 31 (Variables libres) *Les variables libres sont définies par :*

$$FV(x) = \{x\} \qquad FV(!M) = FV(\mathsf{a}_i(M)) = FV(M)$$

$$FV(\lambda x.M) = FV(\nu x.M) = FV(\rho x.M) = FV(M) \setminus \{x\}$$

$$FV((M \quad N)) = FV(\langle M, N\rangle) = FV(M) \cup FV(N)$$

$$FV(\mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N) = (FV(M) \cup FV(N)) \setminus \{x, y\}$$

Equivalences Les équivalences exhibent les différences entre les trois lieurs de λ_{GS}.

Définition 32 (Equivalence de termes)

- *Les lieurs λ et ν sont α-equivalents :*

$$\lambda x.M =_\alpha \lambda z.M[x := z]$$
$$\nu x.M =_\alpha \nu z.M[x := z]$$

  avec $z \notin FV(M)$
- *Les lieurs ν et ρ commutent :*

$$\rho\ell.\rho\ell'.M =_\xi \rho\ell'.\rho\ell.M$$
$$\nu x.\nu y.M =_\xi \nu y.\nu x.M$$

– *Le lieur ρ peut étendre son champ lexical (scope extrusion) :*

$$C[\rho\ell.M] =_\sigma \rho\ell.C[M]$$

pour tout contexte $C[\cdot]$ et $\ell \notin C[\cdot]$.
– *Les lieurs ρ et ν ont une sorte d'η-équivalence :*

$$\begin{array}{ll} \rho o.M =_\gamma M & o \notin M \\ \nu x.M =_\gamma M & w \notin M \end{array}$$

On notera $\div$ l'union de $=_\alpha$, $=_\sigma$, $=_\xi$ et $=_\gamma$.

La scope extrusion est le mécanisme principal de λ_{GS} pour gérer la non-compositionalité. En effet, dans $\rho\ell.M$, ℓ fait référence à un objet physique concret, donc une action à une adresse ℓ peut tout à fait avoir des effets de bord allant plus loin que le champ lexical de $\rho\ell.M$.

C'est typiquement le cas quand on considère un calcul quantique dans le sens où dans :

$$\rho\ell.\langle\rho\ell_1.\rho\ell_2.M, \rho\ell_3.\mathfrak{Cnot}(\langle\ell, \ell_3\rangle)\rangle$$

où $\mathfrak{Cnot}$ est la négation conditionnelle. Le problème est le suivant : si ℓ, ℓ_1 et ℓ_2 sont enchevêtrés dans M, alors une négation conditionnelle sur ℓ_1, ℓ_3 peut enchevêtrer ℓ_3 avec ℓ_1, ℓ_2 qui sont "hors de portée". L'idée fondamentale est qu'une fois qu'un objet s'incarne dans alors cet objet peut avoir influencer, ou être influencé, par n'importe quel autre objet physique faisant partie de cet environnement global.

D'un autre côté le lieur ν ne fait que référer un pointeur abstraite : il n'y a pas encore de réalité physique derrière ce nom. Il s'agit d'une abstraction, purement intellectuelle, sur une ressource physique. Considérons le programme suivant :

$$M1 = (\lambda x.\langle x, x\rangle \quad \nu y.y)$$

on ne veut pas qu'il soit équivalent à :

$$M2 = \nu y.(\lambda x.\langle x, x\rangle \quad y)$$

car l'éxécution de $M1$ crée deux ressources différentes, alors que celle de $M2$ ne crée qu'une ressource. D'où l'absence de scope extrusion concernant le lieur ν.

Sémantique opérationnelle L'état global est une mathématisation d'un état physique. On utilisera pour cela une fonction d'état, $\mathcal{S} : \mathfrak{S}$, dépendante de l'artefact physique utilisé et qui est la contrepartie mathématique de l'objet physique considéré (brin d'ADN, bit quantique, cellule mémoire etc.).

On utilise $\emptyset$ pour dénoter l'état qui n'est défini nul part. Les liens entre les termes de λ_{GS} et l'état global se fait par l'intermédiaire d'une fonction liante qui associe les adresses du terme (donc des termes de λ_{GS}) aux entiers naturels (les adresses des entités physiques). Par exemple dans le cadre d'un système quantique à la λ_{GS}, l'artefact physique peut être un tableau de bits quantique. Une adresse ℓ est associée à un bit quantique (par exemple $\mathcal{K}(\ell) = 12$ indique que ℓ est le 12ème bit quantique du tableau de bits quantiques).

Un système à la λ_{GS} est défini relativement par rapport à un ensemble d'*actions* que l'on peut appliquer à l'état global. Par exemple dans le cas d'une mémoire d'ordinateur, on considère classiquement trois actions pour sa manipulation : l'allocation, la lecture et la mise à jour d'une case mémoire dans laquelle on peut enregistrer une information.

Chaque action $\mathtt{a}_i$, a un effet de bord sur l'artefact physique et peut rendre une valeur (lecture d'une case mémoire, mesure d'un bit quantique) qui peut être utilisée dans la suite du calcul. Donc, si n_i est l'arité de $\mathtt{a}_i$, on doit fournir deux fonctions :

$$\mathcal{F}_i^{\mathcal{D},\mathcal{K}} : (\mathcal{T}^{n_i} \times (\mathfrak{S})) \to (\mathfrak{S})$$
$$\mathcal{F}_i^{\mathcal{T},\mathcal{K}} : \mathcal{T}^{n_i} \to \mathcal{T}$$

$\mathcal{F}_i^{\mathcal{D},\mathcal{K}}$ modélise l'effet de bord sur l'artefact physique par une modification de l'état global (donc une modification de la fonction $\mathcal{S}$) et $\mathcal{F}_i^{\mathcal{T},\mathcal{K}}$ dénote la valeur retournée pour la suite du calcul λ_{GS}.

Pour l'évaluation de termes purs, on utilise la β-réduction usuelle du λ-calcul.

Définition 33 (Evaluation fonctionnelle)

$$(\lambda x.M \quad N) \to_\beta M[x := N]$$

où $M[x := N]$ est M dans lequel toutes les occurrences libres de x ont été remplacées par N.

Si $f : A \to B$ est une fonction, on note $f \uplus \{x \mapsto v\}$ la fonction g telle que $g(y) = f(y)$ pour tout $y \neq x$ et $g(x) = v$.

Définition 34 (Relation de calcul de λ_{GS}) *La relation de calcul,$\rightsquigarrow$, entre triplets de la forme $[\mathcal{S}, \mathcal{K}, M]$ est définie par :*

– *Fonction : si $M \to_\beta M'$:*

$$[\mathcal{S}, \mathcal{K}, M] \rightsquigarrow [\mathcal{S}, \mathcal{K}, M']$$

– *Action :*

$$[\mathcal{S}, \mathcal{K}, (a_i \quad \overrightarrow{M})] \rightsquigarrow [\mathcal{F}_i^{\mathcal{D},\mathcal{K}}(\overrightarrow{M}, \mathcal{S}), \mathcal{K}, \mathcal{F}_i^{\mathcal{T},\mathcal{K}}(\overrightarrow{M})]$$

– *Réalisation :*

$$[\mathcal{S}, \mathcal{K}, !\nu x.M] \rightsquigarrow [\mathcal{S}', \mathcal{K}', \rho o.(M[x := o])]$$

avec $\mathcal{S}'$ et $\mathcal{K}'$ les fonctions d'états et de liaison mises à jour selon le calcul considéré (o sera lié à une nouvelle valeur distinguée prédéfinie et dépendante du calcul considéré).

D'autre part l'idée de ce calcul est que le programmeur n'a jamais à manier le lieur ρ. Ce dernier apparaitra au cours du calcul par concrétisation. Nous donnons donc la défintion suivante

Définition 35 (Termes naturels) *Les termes naturels sont inductivements définis par :*

– *Tout terme issu de la grammaire*

$$P ::= x \mid \ell \mid \lambda x.M \mid (M\ \ N) \mid \langle M, N\rangle \mid \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N \mid \nu x.M \mid !M \mid \mathsf{a}_i$$

est un terme naturel.

– *Si N est un terme naturel et $[\mathcal{S}, \mathcal{K}, N] \rightsquigarrow [\mathcal{S}', \mathcal{K}', M]$ alors M est un terme naturel*

En tenant compte de ces équivalences, une notion de forme standard se dégage naturellement. On peut voire cette forme standard comme un terme distingué pour chaque classe d'équivalence de $\doteqdot$ (cf. def 32).

Fait 2 (Forme standard) *Pour chaque terme naturel M, il existe $N \doteqdot M$ tel que $N \equiv \rho\ell_1.\rho\ell_2.\ldots.\rho\ell_n.N_\rho$ avec $\{\ell_1, \ldots, \ell_n\} \subseteq FV(N_\rho)$ et N_ρ n'ayant pas de sous-terme de la forme $\rho\ell.N'$.*

Dans la suite nous travaillerons implicitement modulo $\doteqdot$. C'est à dire que nous supposerons que les termes sont mis sous forme standard. Donc, quand nous considérons un terme t ayant une forme différente de $\rho\ell.M$, cela signifie de façon implicite que t ne contient pas de lieur ρ. De plus quand on écrit $\rho\ell.M$ cela signifie que ℓ apparait dans M.

Confluence dans λ_{GS} Etant donnée la nature impérative de ce calcul il est clair que de façon générale λ_{GS} risque de ne pas être confluent. En fait λ_{GS}, même en considérant le domaine et les actions les plus simples possibles s'avère non confluent. Considérons le domaine ne contenant qu'un habitant

$\mathcal{D} = \{\mathsf{d}\}$, et aucune action. En alternant β-réduction et concrétisation on obtient une paire critique. Commençons par β-réduire :

$$\begin{array}{rcl}[\emptyset, \mathcal{K}, (\lambda x.(x \;\; x) \;\; !\nu y.y)] & \rightsquigarrow & [\emptyset, \emptyset, (!\nu y.y \;\; !\nu y.y)] \\ & \rightsquigarrow & [\{1 \mapsto \mathsf{d}\}, \{o \mapsto 1\}, \rho o.(!\nu y.y \;\; o)] \\ & \rightsquigarrow & [\{1 \mapsto \mathsf{d}, 2 \mapsto \mathsf{d}\}, \{o \mapsto 1, o' \mapsto 2\}, \rho o'.\rho o.(o' \;\; o)]\end{array}$$

Si sur le même terme nous commençons par la κ-réduction :

$$\begin{array}{rcl}[\emptyset, \emptyset, (\lambda x.(x \;\; x) \;\; !\nu y.y)] & \rightsquigarrow & [\{1 \mapsto d\}, \{o \mapsto 1\}, \rho o.(\lambda x.(x \;\; x) \;\; o)] \\ & \rightsquigarrow & [\{1 \mapsto d\}, \{o \mapsto 1\}, \rho o.(o \;\; o)]\end{array}$$

3.2.2 λ_{GS} pour le calcul quantique

Nous allons développer un langage fonctionnel quantique en nous basant sur λ_{GS}. Cela permet de donner un typage plus flexible, et assurant toujours l'impossibilité du clonage d'un bit quantique, que les propositions de la littérature qui sont basées sur un typage linéaire strict (notamment [SV05a]). Ce système de typage repose fortement sur les nouveaux lieurs qui permettent une distinction syntaxique entre bits quantiques concrets et abstraits. Il implique cependant une stratégie d'évaluation non standard.

Définition de λ^Q_{GS}

λ^Q_{GS} est un langage de programmation quantique basé sur λ_{GS}. Il s'agit d'un langage de programmation fonctionnel pour ordinateur quantique basé sur le modèle QRAM [Kni96].

L'artefact physique de λ^Q_{GS} est un tableau de bits quantiques. n bits quantiques, ou qubits pour abbréger, sont representés comme un vecteur normalisé de l'espace de Hilbert noté $\otimes_{i=1}^{n} C^2$, en utilisant la notation braket : $|\varphi\rangle$.

Les *actions quantiques* sont les portes unitaires classiques suivantes : $\mathfrak{Cnot}, \mathfrak{T}, \mathfrak{H}$ (qui sont respectivement la négation contrôlée, la phase and la transformation d'Hadamard, voir [NC00] pour la définition précise de ces portes quantiques) et la mesure $\mathfrak{M}$. λ^Q_{GS} requiert de plus des booléens $\mathbf{0}$ et $\mathbf{1}$ pour représenter le résultat de la mesure.

La mesure quantique étant probabiliste, l'action $\mathfrak{M}$ l'est aussi. Cependant nous ne nous focaliserons pas sur cet aspect : notre objectif est de définir un système de typage λ^Q_{GS}. Les fonctions définissant les actions unitaires sont définies comme suit :

$$\mathcal{F}_{\mathfrak{T}}^{|\varphi\rangle,\mathcal{K}}(q) = \mathfrak{T}_{\mathcal{K}_q}(|\,\varphi\rangle) \qquad \mathcal{F}_{\mathfrak{H}}^{|\varphi\rangle,\mathcal{K}}(q) = \mathfrak{H}_{\mathcal{K}_q,\mathcal{K}(q')}(|\,\varphi\rangle)$$

$$\mathcal{F}_{\mathfrak{Cnot}}^{|\varphi\rangle,\mathcal{K}}(\langle q,q'\rangle) = \mathfrak{Cnot}_{\mathcal{K}_q}(|\,\varphi\rangle) \quad \mathcal{F}_{\mathfrak{T}}^{\mathcal{T},\mathcal{K}}(q) = \mathcal{F}_{\mathfrak{H}}^{\mathcal{T},\mathcal{K}}(q) = q$$

$$\mathcal{F}_{\mathfrak{Cnot}}^{\mathcal{T},\mathcal{K}}(\langle q,q'\rangle) = \langle q,q'\rangle$$

avec $\mathfrak{T}_i, \mathfrak{H}_i, \mathfrak{Cnot}_{i,j}$ qui sont respectivement les portes de phase, d'Hadamard et de négation conditionnelle appliquées sur les bits quantiques i et j (voir [NC00]).

Soit $|\,\varphi\rangle = \alpha\,|\,\varphi_O\rangle + \beta\,|\,\varphi_1\rangle$ un vecteur normalisé, $|\,0\rangle$ et $|\,1\rangle$ étants les ièmes bits quantiques et

$$|\,\varphi_O\rangle = \Sigma_i \alpha_i\,|\,\phi_i^0\rangle \otimes |\,0\rangle \otimes |\,\psi_i^0\rangle \qquad |\,\varphi_1\rangle = \Sigma_i \beta_i\,|\,\phi_i^1\rangle \otimes |\,1\rangle \otimes |\,\psi_i^1\rangle$$

alors on définit $\mu_0 =|\ \alpha\ |^2$ et $\mu_1 =|\ \beta\ |^2$. On utilise $=_p$ pour définir les fonctions probabilistes : $f(x) =_p y$ signifiant que $f(x)$ retourne y avec la probabilité p.

$$\mathcal{F}_{\mathfrak{T}}^{\alpha|\varphi_O\rangle+\beta|\varphi_1\rangle,\mathcal{K}}(q) =_{\mu_0} 0 \qquad \mathcal{F}_{\mathfrak{T}}^{\alpha|\varphi_O\rangle+\beta|\varphi_1\rangle,,\mathcal{K}}(q) =_{\mu_1} 1$$

$$\mathcal{F}_{\mathfrak{T}}^{\alpha\mathcal{D}_O+\beta|\varphi_1\rangle,\mathcal{K}}(q) = \alpha\,|\,\varphi_O\rangle \qquad \mathcal{F}_{\mathfrak{T}}^{\alpha|\varphi_O\rangle+\beta|\varphi_1\rangle,\mathcal{K}}(q) = \beta\,|\,\varphi_1\rangle$$

La concrétisation met, par défaut, le qbit nouvellement créé à $|\,0\rangle$:

$$[|\,\varphi\rangle, \mathcal{K}, !\nu x.M] \rightsquigarrow [\mathcal{S} \otimes |\,0\rangle, \mathcal{K} \uplus \{o \mapsto n+1\}, \rho o.(M[x := o])]$$

Système de type pour λ_{GS}^{Q}

Les types de λ_{GS}^{Q} sont définis par :

$$\tau ::= \mathbf{B} \mid \mathbf{Q} \mid \tau \to \tau \mid \tau \times \tau \mid \Diamond\tau$$

$\mathbf{B}$ est le type booléen avec les constantes $\mathbf{0}, \mathbf{1}$. Les λ-abstractions seront annotées par le type (système à la Church), par contre, étant donné que les lieurs ρ et ν ne peuvent lier que des bits quantiques il n'y a pas d'annotation de type pour ces lieurs.

$\mathbf{Q}$ est le type des qubits. Nous attirons l'attention sur le fait qu'il n'y a pas de constantes de type $\mathbf{Q}$ dans λ_{GS}^{Q}. Les qubits ne sont manipulés qu'au travers d'adresses et de la fonction de lien. Sinon il y aurait des problèmes de non localité dans les calculs quantiques et de représentation des états enchevêtrés (voir [SV05b] pour plus d'explications à ce sujet). Contrairement à [SV05b] il

n'y a pas de notion primitive de types linéaires (il n'y a pas d'application de fonction linéaire ni de types exponentiels) : la linéarité sera uniquement gérée au travers des lieurs ν and ρ. Le type $\Diamond\tau$ est utilisé pour typer les ν-redex.

Les contextes de typage, écrits $\Gamma;\Delta$, sont des listes de couples composés d'une variable et de son type, listes dans lesquelles les variables sont définies de manière univoque. Pour rendre la lecture plus aisée nous découpons les contextes de typage en deux parties. La première partie (Γ) est *classique* alors que la seconde (Δ) est *linéaire*. Par exemple dans le contexte de typage suivant : $x_1 : \tau_1, \ldots, x_n : \tau_n; y_1 : \tau_1, \ldots, y_m : \tau_m$, chaque x_i est classique et peut être réutilisé (grâce à la règle de contraction $[CTN]$) et éliminé à volonté (règle d'affaiblissement [WkgI]), et chaque y_i est linéaire et doit être utilisé exactement une fois mais peut aussi être éliminé (règle d'affaiblissement linéaire [WkgL]).

Définition 36 (Jugement de typage) *Un jugement de typage dans λ^{Q}_{GS} est un n-uplet $\Gamma;\Delta \vdash M : \tau$ qui peut être établi en utilisant les règles de typages définies dans Fig. 3.1 page 50.*

Ce système de typage repose donc sur une distinction entre environnements classique et linéaire. L'objectif est que les bits quantiques concrets soient utilisés de manière linéaire eut égard à l'axiome de no-cloning. Par contre il est possible de cloner des termes dont les qubits sont abstraits (guardés par des lieurs ν). A ce sujet, nous attirons l'attention sur la condition de la règle $[AxI]$. Dans cette règle M ne contient pas de lieur ρ, et comme il n'y a pas de règle pour introduire le type Q, cela ne peut se faire sur un terme clos qu'au travers des lieurs ν et ρ. De plus comme il n'y a pas de concrétisation dans le terme ce dernier ne peut pas conduire par réductions à un terme contenant des lieurs ρ et donc in fine a des qubits concrets. D'un autre côté la règle $[AxL]$ montre qu'on peut introduire linéairement des variables de n'importe quel type.

La condition de la règle $[AxI]$ est réutilisée pour typer les règles éliminant les variables des contextes telles que $[\rightarrow II]$, $[\times ELI]$, $[\times EIL]$ et $[\times EII]$ (en fait on pourrait la retirer pour ces dernières règles car si cette condition n'est pas vérifiée alors on ne pourra pas inférer la prémisse, ces conditions sont donc redondantes). Elle sert à vérifier que le type d'une variable est bien classique.

Cette condition conduit à la définition :

Définition 37 (Type classique) *Un type τ est dit classique s'il existe un terme M sans lieur ρ et sans ! tel que $.;. \vdash M : \tau$.*

Par opposition si un type n'est pas classique il sera dit linéaire.

$$\frac{\cdot;\cdot \vdash M : \tau \quad !\notin M}{y:\tau;\cdot \vdash y:\tau}[AxI] \qquad \frac{}{\cdot; y:\tau \vdash y:\tau}[AxL]$$

$$\frac{\Gamma_1, y:\sigma', x:\sigma, \Gamma_2; \Delta \vdash M:\tau}{\Gamma_1, x:\sigma, y:\sigma', \Gamma_2; \Delta \vdash M:\tau}[ExI] \qquad \frac{\Gamma; \Delta_1, y:\sigma', x:\sigma, \Delta_2 \vdash M:\tau}{\Gamma; \Delta_1, x:\sigma, y:\sigma', \Delta_2 \vdash M:\tau}[ExL]$$

$$\frac{\Gamma;\Delta \vdash M:\tau}{\Gamma, x:\sigma;\Delta \vdash M:\tau}[WkgI] \qquad \frac{\Gamma;\Delta \vdash M:\tau}{\Gamma;\Delta, x:\sigma \vdash M:\tau}[WkgL]$$

$$\frac{\Gamma, x:\tau, y:\tau;\Delta \vdash M:\sigma}{\Gamma, z:\tau;\Delta \vdash M[x:=z;y:=z]:\sigma}[CTN]$$

$$\frac{\Gamma, x:\sigma;\Delta \vdash M:\tau \quad \cdot;\cdot \vdash N:\sigma \quad !\notin N}{\Gamma;\Delta \vdash \lambda x{:}\sigma.M:\sigma\to\tau}[\to II] \quad \frac{\Gamma;\Delta, x:\sigma \vdash M:\tau}{\Gamma;\Delta \vdash \lambda x{:}\sigma.M:\sigma\to\tau}[\to IL]$$

$$\frac{\Gamma;\Delta_1 \vdash M:\sigma\to\tau \quad \Gamma;\Delta_2 \vdash N:\sigma}{\Gamma;\Delta_1,\Delta_2 \vdash (M \quad N):\tau}[\to E]$$

$$\frac{\Gamma;\Delta_1 \vdash M:\tau \quad \Gamma;\Delta_2 \vdash N:\sigma}{\Gamma;\Delta_1,\Delta_2 \vdash \langle M,N\rangle:\tau\times\sigma}[\times I]$$

$$\frac{\Gamma;\Delta_1 \vdash N:\tau_1\times\tau_2 \quad \Gamma;\Delta_2, x:\tau_1, y:\tau_2 \vdash M:\sigma}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ \langle x,y\rangle = N\ \mathsf{in}\ M:\sigma}[\times ELL]$$

$$\frac{\Gamma;\Delta_1 \vdash N:\tau_1\times\tau_2 \quad \Gamma, x:\tau_1;\Delta_2, y:\tau_2 \vdash M:\sigma \quad \cdot;\cdot \vdash P:\tau_1 \quad !\notin P}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ \langle x,y\rangle = N\ \mathsf{in}\ M}[\times ELI]$$

$$\frac{\Gamma;\Delta_1 \vdash N:\tau_1\times\tau_2 \quad \Gamma, y:\tau_2;\Delta_2, x:\tau_1 \vdash M:\sigma \quad \cdot;\cdot \vdash P:\tau_2 \quad !\notin P}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ \langle x,y\rangle = N\ \mathsf{in}\ M}[\times EIL]$$

$$\frac{\Gamma;\Delta_1 \vdash N:\tau_1\times\tau_2 \quad \Gamma, x:\tau_1, y:\tau_2;\Delta_2 \vdash M:\sigma \quad \begin{array}{c}\cdot;\cdot \vdash P:\tau_2 \quad !\notin P\\ \cdot;\cdot \vdash Q:\tau_1 \quad !\notin Q\end{array}}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ \langle x,y\rangle = N\ \mathsf{in}\ M}[\times EII]$$

$$\frac{\Gamma;\Delta, x:\mathbf{Q} \vdash M:\tau}{\Gamma;\Delta \vdash \nu x.M:\Diamond\tau}[\nu I]$$

$$\frac{\Gamma;\Delta \vdash M:\Diamond\tau}{\Gamma;\Delta \vdash !M:\tau}[\nu E]$$

$$\frac{\Gamma;\Delta, \ell:\mathbf{Q} \vdash M:\tau}{\Gamma;\Delta \vdash \rho\ell.M:\tau}[\rho I]$$

FIGURE 3.1 – Règles de typage de λ^{Q}_{GS}

On peut remarquer qu'il n'y a pas de règle d'élimination pour le lieur ρ, ce rôle est rempli par la pseudo η-équivalence sur ρ (cf. définition 32). Cela n'affecte pas la stabilité du typage par la réduction car il n'y a pas d'effet sur le type du terme.

Considérons par exemple le programme suivant :

$$cf \stackrel{def}{=} \nu x.\mathfrak{M}(\mathfrak{H}(x))$$

alors $.;. \vdash cf : \Diamond\mathbf{B}$ est dérivable. Donc il est possible d'utiliser $x_{cf} : \Diamond\mathbf{B}$ dans le contexte de typage classique. C'est intéressant car cf est la formalisation d'un tirage aléatoire d'une pièce de monnaie, c'est un morceau de programme qu'on pourrait avoir à utiliser à différents endroits du programme (et donc il faudrait pouvoir le dupliquer). Cet exemple simple n'est pas possible à typer dans des systèmes où les qubits doivent être utilisés de manière linéaire (par exemple [SV05a]).

Dans la définition 34 nous avons défini de manière général les réductions de calcul pour un système λ_{GS}. Il faut adapter cette définition pour obtenir un système vérifiant la propriété de conservation du typage. En effet, si on ne fixe pas une stratégie de réduction pour λ^{Q}_{GS} on peut se retrouver face à des termes suivants :

$$(\lambda y{:}\mathbf{B}.\langle y, y\rangle \;\; (m \;\; o))$$

où m est de type $\mathbf{Q} \to \mathbf{B}$ (on peut penser à la mesure voir même à une fonction constante du type $\lambda y.\mathbf{0}$). Le problème est que si l'on réduit ce redex on se retrouve avec plusieurs occurrences de o et le terme ne sera plus typable. On remarquera qu'essentiellement il n'y a que la mesure qui fasse passer d'un type linéaire à un type classique sinon il faut passer par une projection du qbit du style fonction constante. Il ne parait donc pas déraisonnable de bloquer la β-réduction en attendant le résultat de la mesure (ou de la projection). Par contre on peut noter que

$$(\lambda y{:}\mathbf{B}.\langle y, y\rangle \;\; !\nu x.(m \;\; x))$$

est quant à lui réductible car le qbit x est abstrait.

Ces remarques conduisent aux définitions suivantes :

Définition 38 (Termes réductibles) *Etant donné un environnement de typage $\Gamma;\Delta$ un terme N est dit $\Gamma;\Delta$-réductible si pour chaque $\Gamma;\Delta \vdash P : \tau$ avec τ un type linéaire et $C[P] = N$ pour un certain contexte $C[\cdot]$, alors pour chaque $x \in FV(P)$ ayant un type linéaire, alors il existe des contextes $C'[\cdot], C''[\cdot]$ tels que $C'[\nu x.C''[P]] = N$.*

Autrement dit un terme est réductible si pour chacun de ses sous-termes ayant un type linéaire ce dernier ne comporte que des variables de type linéaire qui sont liées par ν.

Définition 39 (Réduction guardée) *Soit Gamma*; Δ *un environnement de typage, et considérons le redex* $(\lambda x{:}\tau.M\ \ N)$. *Ce rédex se réduit en* $M[x := N]$ *par réduction gardée (notée* $\rightarrow^Q_\beta$*) si et seulement si :*

- *Soit* τ *est un type linéaire,*
- *soit* τ *est un type classique et* N *est un terme* $\Gamma;\Delta$*-réductible.*

Définition 40 (Réduction dans λ^Q_{GS}) *La relation de calcul de* λ^Q_{GS} *notée* $\rightsquigarrow^Q$ *est définie par :*

- *Fonction : si* $M \rightarrow^Q_\beta M'$ *:*

$$[\mathcal{S}, M] \rightsquigarrow^Q [\mathcal{S}, M']$$

  La réduction des let*-expressions est traitée comme une double* β*-réduction gardée.*
- *Actions :*
  - $[\mathcal{S}, \mathfrak{Cnot}(\langle o, o'\rangle)] \rightsquigarrow^Q [\mathcal{S}', \langle o, o'\rangle]$
  - $[\mathcal{S}, \mathfrak{T}(o)] \rightsquigarrow^Q [\mathcal{S}', o]$
  - $[\mathcal{S}, \mathfrak{H}(o)] \rightsquigarrow^Q [\mathcal{S}', o]$
  - $[\mathcal{S}, \mathfrak{M}(o)] \rightsquigarrow^Q [\mathcal{S}', \frac{\mathbf{0}}{\mathbf{1}}]$
- *Réalisation :*

$$[\mathcal{S}, !\nu x.M] \rightsquigarrow [\mathcal{S} \otimes |0\rangle, \rho o.(M[x := o])]$$

  avec o *un nom frais.*

Théorème 7 (Stabilité du typage par réduction) *Soit* M *tel que* $\Gamma;\Delta \vdash M : \tau$*, alors si* $[\mathcal{S}, M] \rightsquigarrow^Q [\mathcal{S}, M']$ *on a* $\Gamma;\Delta \vdash M' : \tau$

3.3 Logique d'enchevêtrement / séparabilité

Nous allons présenter une analyse statique de l'enchevêtrement (et de la propriété duale de séparabilité) des bits quantiques au cours d'un calcul. L'idée est d'adapter un résultat classique, celui de [BHY05] qui était une logique de Hoare (voir [Hoa69]) dans un cadre avec pointeurs, dans un contexte quantique. Il existe en effet de nombreux points communs entre les problèmes soulevés par l'enchevêtrement et les alias. Il y a notamment cette idée qu'en agissant sur un nom on peut agir sur un autre nom (quand les pointeurs sont

aliasés) qui est similaire au fait qu'agir sur un bit quantique peut avoir une action sur un bit qui est enchevêtré avec lui.

Nous cherchons à établir des jugements de la forme suivante :

$$\{C\}M :^{\Gamma;\Lambda;\Delta;\tau} u\{C'\}$$

C est traditionellement appelé la précondition, C' la post-condition, M est le sujet et u est une ancre (il s'agit du nom que l'on utilisera dans C' pour dénoter la valeur de M). Intuitivement cela signifie que si C est satisfaite, alors après l'évaluation de M, dont la valeur est dénotée par u dans C', C' est satisfaite. $\Gamma;\Lambda$ étant le contexte de typage de M, τ le type de M et Δ est le contexte de typage des ancres : ce contexte est utilisé pour avoir des assertions bien typées. Dans notre contexte particulier, les pré/post-conditions seront des formules décrivant des propriétés sur l'enchevêtrement/la séparabilité des bits quantiques de la mémoire quantique. Typiquement on aura des jugements tels que :

$$\{u \leftrightarrow v\}M :^{\Gamma;\Lambda;\Delta;\tau} u\{w \leftrightarrow x\}$$

dont la signification intuitive est la suivante : si u et v sont des bits quantiques enchevêtrés alors l'éxécution de M conduira à une mémoire quantique dans laquelle w et x seront enchevétrés.

Donc les assertions permettent d'établir si deux bits quantiques sont enchevêtrés ou séparables et comme ces propriétés ne sont pas calculables – car on peut trivialement les réduire au problème de l'arrêt en ajoutant par exemple $\mathfrak{Cnot}(q_i, q_j)$ comme dernière instruction d'un programme de telle manière à ce que q_i et q_j sont enchevêtrés ssi le calcul s'arrête – les assertions sont des approximations sûres. Si une assertion établit que deux bits quantiques sont séparables alors ils le sont vraiment. Par contre si l'assertion établit que les deux bits quantiques sont enchevêtrés alors il est possible qu'ils ne le soient pas.

3.3.1 λ_L^Q un langage fonctionnel pour la programmation quantique

Nous utilisons une variante du λ-calcul de Selinger et Valiron [SV05a] comme language de programmation avec un ensemble fini de portes quantiques (sans perte d'expressivité). Nous allons aussi supposer pour cette étude que le nombre de bits quantiques est fixé, il n'y a donc pas d'opérateur ν (et de concrétisation associée comme c'était le cas dans la section 3.1) permettant de créer de nouveaux bits quantiques au cours du calcul. Nous supposerons donc dans la suite que le nombre de bits quantiques est fixe et égal à un certain entier n.

Définition 41 (Termes et types) *λ_L^Q Les termes et les types sont définis par :*

$$\begin{array}{rcl} M & ::= & x \mid q_i \mid \mathbf{0} \mid \mathbf{1} \mid \lambda x{:}\sigma.N \mid (M \;\; N) \mid \\ & & \langle M, N\rangle \mid \\ & & \mathsf{let}\ \langle x, y\rangle : \sigma \otimes \tau = M\ \mathsf{in}\ N \\ & & \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P \mid \\ & & \mathsf{meas} \mid \mathfrak{Cnot} \mid \mathfrak{H} \mid \mathfrak{T} \\ \sigma & ::= & \mathbf{B} \mid \mathbf{Q} \mid \sigma \to \tau \mid \sigma \otimes \tau \end{array}$$

avec x dénotant les noms d'éléments d'un ensemble dénombrable de variables. q_i, avec $i \in \{1..n\}$ sont les noms constants qui sont utilisés pour pointer les bits quantiques concrets. $\mathbf{0}, \mathbf{1}$ sont les constantes booléennes habituelles. Les actions quantiques sont respectivement la mesure et les trois portes quantiques de négation conditionnelle, Hadamard et Phase.

Il n'y a que deux types de bases $\mathbf{B}$ pour les bits et $\mathbf{Q}$ pour les bits quantiques.

On pourra noter que si les q_i sont des constantes, il est possible d'avoir des variables de type bit quantique dans λ_L^Q. Par exemple :

$$(\lambda x{:}\mathbf{Q}.N \;\; \mathsf{if}\ M\ \mathsf{then}\ q_1\ \mathsf{else}\ q_2)$$

Après réduction x va finalement devenir soit q_1 soit q_2 dans N, suivant l'évaluation de M.

Enfin, pour ne pas avoir d'utilisation redondante de qubits ces constantes doivent être utilisées de manière linéaire d'où la nécessité d'un environnement quantique pour contrôler la bonne formation des termes.

Définition 42 (Contextes et jugements de typages) *Les contextes classiques sont définis par : $\Gamma ::= \cdot \mid \Gamma, x : \sigma$ avec $\mathbf{Q} \notin \sigma$.*

Les contextes linéaires sont définis par : $\Lambda ::= \cdot \mid \Lambda, x : \sigma$ avec $\mathbf{Q} \in \sigma$.

Les contextes quantiques sont un sous ensemble de $\{1, \ldots, n\}$ et dénotés par κ.

Les jugements de typage sont de la forme : $\Gamma; \Lambda; \kappa \vdash M : \sigma$ et doit être lu de la façon suivante : étant donné le contexte de typage classique Γ, le contexte de typage linéaire Λ et le contexte quantique κ, le terme M est bien formé de type σ.

Quand on écrit $\Gamma, x : \sigma; \Delta$ (resp. $\Gamma; \Lambda, y : \tau$) on suppose implicitement que x (resp. y) n'apparaît pas dans $\Gamma; \Lambda$ (resp. $\Gamma; \Lambda$). De même quand on écrit Γ_1, Γ_2 (resp. Λ_1, Λ_2 ou κ_1 et κ_2) on suppose que Γ_1 et Γ_2 (resp. Λ_1 et Λ_2 ou κ_1 et κ_2)vsont disjoints.

Les règles de typage de λ_L^Q sont standard et très proches de celles de DILL [Bar96] avec les différences suivantes : il n'y a pas de règle de dérilection, principalement parce que λ_L^Q n'est sensible aux ressources qu'en ce qui concerne les bits quantiques, pour la même raison il n'y a pas de règle de promotion. Par contre cela implique d'introduire la notion de type intuitionniste/quantique.

Définition 43 (Types intuitionistes/quantiques) *Un type est dit intuitionniste s'il est engendrable à partir de la grammaire suivante :*

$$\sigma ::= \mathbf{B} \mid \mathbf{Q} \mid \sigma \to \tau \mid \sigma \otimes \tau$$

dualement un type non intuitionniste sera dit quantique.

Nous donnons les règles de typages de λ_L^Q en figure Fig. 3.2 page 66. On remarquera que l'élimination de la flèche implique une séparation des contextes linéaires et est linéaire. Pour une β-réduction classique il faut passer par un let qui permet d'assurer que l'argument passé en paramètre réel ne comporte pas d'éléments linéaires : le contexte linéaire pour typer l'argument doit être vide et ce dernier d'un type intuitioniste. Ceci permet d'assurer le lemme de substitution (cf. [Bar96]). D'un point de vue typage, les portes quantiques sont typées comme la fonction identité ce qui est normal car du point de vue des termes elles se comportent comme tel (voir la sémantique opérationnelle donnée dans la définition 46 page 56).

Définition 44 (λ_L^Q-état) *Soit $\Gamma;\Lambda;\kappa \vdash M : \sigma$. Un λ_L^Q-état est un couple $[|\varphi\rangle, M]$ avec $|\varphi\rangle$ un vecteur normalisé d'un espace de Hilbert $\mathbb{C}^{2^n}$ et M un terme λ_L^Q.*

Un exemple de λ_L^Q-état de taille $n = 2$ est :

$$[|\varphi\rangle, (\lambda q{:}\mathbf{Q}.\mathsf{if}\ (\mathsf{meas}\ \ q_1)\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ (\mathsf{meas}\ \ (\mathfrak{T}\ \ q))\ \ q_2)]$$

avec $|\ \varphi\rangle = \frac{1}{\sqrt{2}}(|\ \mathbf{1}\rangle + |\ \mathbf{0}\rangle) \otimes (\frac{2}{3}\ |\ \mathbf{1}\rangle + \frac{\sqrt{5}}{3}\ |\ \mathbf{0}\rangle)$ q_1 est le bit quantique dénoté par $\frac{1}{\sqrt{2}}(|\ \mathbf{1}\rangle + |\ \mathbf{0}\rangle)$ et q_2 est celui représenté par $\frac{2}{3}\ |\ \mathbf{1}\rangle + \frac{\sqrt{5}}{3}\ |\ \mathbf{0}\rangle$. Ce programme mesure q_1, et suivant le résultat de la mesure, qui est soit $\mathbf{0}$ soit $\mathbf{1}$ avec probabilité 1/2, il mesure q_2 ou rend $\mathbf{0}$. Donc la valeur calculée par ce morceau de code est $\mathbf{0}$ avec une probabilité de 7/9 et $\mathbf{1}$ avec une probabilité de 2/9.

Définition 45 (Valeurs) *Les valeurs de λ_L^Q sont définies par : $U, V ::= x \mid \mathbf{0} \mid \mathbf{1} \mid q_i \mid \lambda x{:}\sigma.M \mid \langle V, V\rangle \mid (F\ \ V)$ avec F un des opérateurs suivants $\pi_i, \mathfrak{T}, \mathfrak{H}, \mathfrak{Cnot}, \mathsf{meas}$*

Définition 46 (Réductions quantiques) *On définit une relation de réduction probabiliste entre λ_L^Q-états, $[|\varphi\rangle, M] \to_p [|\varphi'\rangle, M']$, qui doit être lue de la façon suivante : $[|\varphi\rangle, M]$ se réduit en $[|\varphi'\rangle, M']$ avec probabilité p.*

Les règles de réduction sont données en figure Fig. 3.3 page 67. Nous ne donnons que les règles concernant le quantique. En ce qui concerne la partie fonctionnelle les réductions sont standard (voir par exemple [Pie02]) pour un calcul en appel par valeur et laissent les probabilité inchangées.

Dans les règles $[MET]$ et $[MEF]$, soit $|\varphi\rangle = \alpha\,|\varphi_{\mathbf{1}}\rangle + \beta\,|\varphi_{\mathbf{0}}\rangle$ normalisé avec

$$|\varphi_{\mathbf{0}}\rangle = \textstyle\sum_i = 1^n \alpha_i\,|\phi_i^{\mathbf{0}}\rangle \otimes |\mathbf{0}\rangle \otimes |\psi_i^{\mathbf{0}}\rangle$$
$$|\varphi_{\mathbf{1}}\rangle = \textstyle\sum_i = 1^n \beta_i\,|\phi_i^{\mathbf{1}}\rangle \otimes |\mathbf{1}\rangle \otimes |\psi_i^{\mathbf{1}}\rangle$$

où $|\mathbf{0}\rangle$ et $|\mathbf{1}\rangle$ dénote le ième bit quantique.

Un calcul sur un terme de λ_L^Q part d'un état inital $|\varphi_{init}\rangle$ qui est défini par convention comme suit : $|\varphi_{init}\rangle = |\mathbf{1}\rangle \overbrace{\otimes \ldots \otimes}^{n-1} |\mathbf{1}\rangle$

Proposition 1 (Stabilité du typage par réduction) *Soit $\Gamma; \Lambda; \kappa \vdash M : \tau$ et $[\varphi, M] \to_p [\varphi', M']$, alors $\Gamma; \Lambda; \kappa \vdash M' : \tau$*

3.3.2 Assertions d'enchevêtrement

Définition 47 *Les termes et les assertions sont définis par :*

$$\begin{array}{rcl} e & ::= & u \mid q_i \mid \mathbf{0} \mid \mathbf{1} \mid \langle e, e'\rangle \mid \pi_i(e) \\ C & ::= & \mathbf{0} \mid \mathbf{1} \mid u \leftrightarrow v \mid \|e \mid \not\|e \\ & & \mid C \vee C' \mid C \wedge C' \mid C \implies C' \\ & & \mid \forall u : \sigma.C \mid \exists u : \sigma.C \\ & & \mid \{C\}e_1 \bullet e_2 = e_3\{C'\} \end{array}$$

où u, v sont des noms d'ancres pris dans un ensemble dénombrables.

Chaque sous-terme d'un programme est identifié dans une assertion par une ancre, qui est simplement un nom. Notons que les noms des bits quantiques sont considérés comme des termes de base.

L'assertion $u \leftrightarrow v$ signifie que le bit quantique u est **peut être** enchevêtré avec v. L'assertion $\{C\}e_1 \bullet e_2 = e_3\{C'\}$ est utilisée pour gérer les fonctions dans le cadre logique. Il s'agit de la formule d'évaluation. e_3 est lié dans C'. En reprenant la terminologie de [BHY05], C, C' sont respectivement appelées des pré/post-conditions internes. L'idée est que l'invocation d'une fonction dénotée par l'ancre e_1 avec l'argument e_2, et si la précondition C est satisfaite

par l'état quantique courant, va s'évaluer et produire un état quantique dans lequel C' sera satisfaite, e_3 étant le nom de la valeur retournée par la fonction (e_3 est un nom lié dans C').

Les autres assertions gardent leurs significations habituelles en logique du premier ordre. T l'assertions toujours vraie. Notons que les lieurs $\forall$ et $\exists$ sont typés. En effet, dans cette logique il faut garder à l'esprit que les noms sont en dernière analyse des représentants de termes, et par là peuvent par exemple dénoter des fonctions. Il y a donc une notion d'assertion bien formée vis à vis des contraintes de typage.

Définition 48 (Typage des assertions)

- $\Gamma; \Lambda; \Delta \vdash t : \tau$ *est un jugement énonçant que le terme logique* t *est bien typé de type* τ *étant donné un contexte de typage d'assertions* $\Gamma; \Lambda; \Delta$. *Les règles pour établir ce jugement sont données dans la figure Fig. 3.4 page 67, dans laquelle* $i \in \{1, 2\}$, *et* $\mathbf{c}$ *sont soit* $\mathbf{0}$ *ou* $\mathbf{1}$.
- *De mêne assertion* C *est bien typée sous un contexte de typage* $\Gamma; \Lambda; \Delta$, *ce que l'on écrit* $\Gamma; \Lambda; \Delta \vdash C$ *si l'on peut établir le jugement en utilisant les règles de donnée en figure Fig. 3.5 page 67 (nous ne donnons que quelques règles pour illustrer cette définition standard sinon et nous nous référons à [PZ09] pour l'ensemble exhaustif), avec* $(\Lambda; \Delta) \setminus u : \sigma$ *est le contexte* $\Lambda; \Delta$ *sans* $u : \sigma$.

On peut classer les règles de typages des assertions suivant deux catégories. Dans la première on intègre les règles qui assurent une utilisation correcte des ancres par rapport aux types des termes qu'elles dénotent. Ce sont les règles $[TAs \leftrightarrow]$ $[TAs\|]$ $[TAs =]$ $[TAs\forall]$ $[TAs\exists]$ et $[TAsEV]$. Dans le second groupe sont regroupées les règles structurelles : $[TAs\neg]$ $[TAs\wedge]$ $[TAs\vee]$, et $[TAs \implies]$.

3.3.3 Sémantique des assertions d'enchevêtrement

Nous allons maintenant formaliser la sémantique intuitive derrière les assertions. Pour cela nous allons abstraire l'état quantique en un état quantique abstrait qui sera une approximation sûre d'à la fois la relation d'enchevêtrement entre les bits quantiques de cet état mais aussi du sous-ensemble des bits quantiques étants dans un état de base. La sémantique des assertions sera définie en relation avec cet état quantique abstrait. De plus nous developpons une sémantique opérationnelle abstraite permettant d'éxécuter un programme de λ_L^Q sur un état quantique abstrait.

Etat quantique abstrait et sémantique opérationnelle abstraite

On notera $S = \{q_1, \ldots, q_n\}$, l'ensemble de bits quantiques. L'état quantique S est comme d'usage décrit par $|\varphi\rangle$, un vecteur normalisé de $\mathbb{C}^{2^n}$.

Définition 49 (Etat quantique abstrait) *Un état quantique abstrait (EQA) sur S est un nuplet $A = (\mathcal{R}, \mathcal{P})$ où $\mathcal{P} \subseteq S$ et $\mathcal{R}$ est une relation d'équivalence partielle sur $(S \setminus \mathcal{P}) \times (S \setminus \mathcal{P})$.*

La relation $\mathcal{R}$ est une relation d'équivalence partielle car elle décrit une approximation de la relation d'enchevêtrement. Il n'y a en effet pas vraiment de sens de parler d'un bit quantique enchevêtré avec lui même. De fait, à cause de l'axiome de clonage on ne peut pas construire de programme $p : \mathbf{Q} \times \mathbf{Q} \to \tau$ requérant deux bits quantiques séparables et construire une terme comme $(p\ \ \langle q_i, q_i \rangle)$ (donc il n'est pas nécessaire de vérifier la réflexivité pour l'EQA).

La classe d'équivalence d'un bit quantique q par rapport à un EQA $A = (\mathcal{R}, \mathcal{P})$ est dénotée par $\overline{q}^A$.

Définition 50 (EQA : adéquation avec l'état quantique) *Soit S dont l'état est représenté par $|\ \varphi\rangle$ et $A = (\mathcal{R}, \mathcal{P})$ un EQA sur S. A est en adéquation (ou plus simplement adéquat) avec $|\varphi\rangle$, ce que l'on écrit $A \models |\varphi\rangle$, ssi pour chaque $x, y \in S$ tels que $(x, y) \notin \mathcal{R}$ alors x, y sont séparables dans $|\varphi\rangle$ et pour chaque $x \in \mathcal{P}$, la mesure de x est déterministe.*

Supposons que $S = \{q_1, q_2, q_3\}$ et $|\varphi\rangle = 1/\sqrt{(}2)(|\mathbf{11}\rangle + |\mathbf{00}\rangle) \otimes |\mathbf{0}\rangle$ alors $A = (\{(q_1, q_2), (q_2, q_1)\}, \{q_3\})$ et $A' = (\{(q_1, q_2), (q_2, q_1), (q_2, q_3), (q_3, q_2), (q_3, q_1), (q_1, q_3)\}, \emptyset)$. On a $A \models |\ \varphi\rangle$, car q_3 est dans un état de base et q_1, q_2 sont séparables de q_3. De même on a également $A' \models |\ \varphi\rangle$. En effet A' est une approximation de l'ensemble des bits quantiques dans un état de base en établissant que tous les bits quantiques sont enchevêtrés deux à deux (même s'il s'avère que q_3 est en fait dans un état de base). D'un autre côté $B = (\{(q_1, q_2), (q_2, q_1)\}, \{q_2, q_3\})$ et $B' = (\emptyset, \{q_3\})$, ne sont pas des états quantiques abstraits adéquats vis à vis de $|\varphi\rangle$. B n'est pas adéquat car q_2 n'est pas dans un état de base, et B' n'est pas adéquat car relativement à l'équivalence partielle de B, q_1, q_2 sont supposés séparables.

Définition 51 (Sémantique opérationnelle abstraite) *La sémantique opérationnelle abstraite d'un terme M tel que $\Gamma; \Lambda \vdash M : \tau$ est une relation entre couples constitués d'un EQA et d'un terme : $[A, M] \to_{\mathcal{A}}^{\Gamma,\Lambda} [A', M']$*

Nous écrirons $\to_{\mathcal{A}}$ à la place de $\to_{\mathcal{A}}^{\Gamma,\Lambda}$ quand le contexte de typage n'a pas d'importance ou peut être inféré du contexte.

En ce qui concerne la partie fonctionnelle, les règles d'évaluations sont les mêmes que pour la définition 46 en remplançant l'état quantique par un EQA. Nous donnons les règles pour les actions quantiques dans la figure 3.6.

Où $\begin{matrix}\mathbf{0}\\\mathbf{1}\end{matrix}$ *est de manière non déterministe soit* $\mathbf{0}$ *soit* $\mathbf{1}$*,* $\mathcal{R} \setminus q_i$ *est la relation d'équivalence telle que si* $(x, y) \in \mathcal{R}$ *et* $x \neq q_i$ *ou bien (ou à comprendre dans un sens exclusif)* $y \neq q_i$ *alors* $(x, y) \in \mathcal{R} \setminus q_i$ *soit* $(x, y) \notin \mathcal{R} \setminus q_i$*, avec* $\mathcal{R} \cdot q_i \leftrightarrow q_j$ *la relation d'équivalence* $\mathcal{R}$ *pour laquelle les classes d'équivalences de* q_i, q_j *ont été fusionnées.*

Comme notre système est normalisant l'ensemble des éxécutions abstraites associées à un programme est fini donc calculable.

Définition 52 (Sémantique abstraite d'un programme) *Soit un EQA* A*, la semantique abstraite d'un programme* $\Gamma; \Lambda; \kappa \vdash M : \tau$ *étant donné* A*, que l'on écrit* $[\![M]\!]_A^{\Gamma;\Lambda}$*, est la réunion des* A' *tels que* $[A, M] \rightarrow_{\mathcal{A}}^* [A', V]$ *où* V *est une valeur.*

Il est important de noter que la sémantique abstraite d'un programme est une sémantique qui collecte toutes les éxécutions abstraites possibles. Il se peut qu'on explore des branches qui ne seront jamais utilisées dans un calcul réel car dans la sémantique opérationnelle la mesure donne un résultat non déterministe. De plus l'objectif de cette sémantique abstraite est juste de collecter ce qui se passe au niveau des classes d'enchevêtrements. Cependant nous avons le résultat suivant qui lie les sémantiques abstraites et concrètes :

Proposition 2 *Supposons que* $A \models |\, \varphi\rangle$ *et* $\Gamma; \Lambda; \kappa \vdash M : \tau$*. Supposons également* $[|\, \varphi\rangle, M] \rightarrow_\gamma^* [|\, \varphi'\rangle, V]$ *alors il existe* $A' \models |\, \varphi'\rangle$ *tel que* $[A, M] \rightarrow_{\mathcal{A}}^* [A', V]$*.*

Ce résultat permet d'établir le lien entre la satisfaction d'une assertion et l'état quantique concret. En effet, la validité des assertions est définie relativement à un EQA qui à son tour est relié à l'état quantique par le résultat précédent.

Sémantique des assertions

La sémantique d'une assertion bien typée est donnée par rapport à un état quantique concret mais se fait par le truchement d'un EQA adéquat. Supposons que $|\, \varphi\rangle \models A$, et $\Gamma; \Lambda; \Delta \vdash C$, alors quand C est satisfaite par une interprétation $\mathcal{I}$, on écrit $\mathcal{M}^{\Gamma;\Lambda;\Delta} \models C$ avec $\mathcal{M}^{\Gamma;\Lambda;\Delta} = \langle A, \mathcal{I}\rangle$. Pour que ce soit le cas il faut vérifier deux propriétés : Est-ce que deux bits quantiques sont dans la même classe d'équivalence ? Est-ce qu'un bit quantique est dans

un état de base ? Le reste des connecteurs est interprété comme de coutume en logique du premier ordre.

Nous déinissons maintenant une équivalence observationnelle abstraite. Cette équivalence observationnelle n'est valide qu'en ce qui concerne l'effet du programme quant aux relations d'intrication (ce n'est donc pas une relation d'équivalence observationnelle au sens traditionnel du terme), en effet elle ne sert qu'à valider des assertions portants sur des relations d'enchevêtrement.

Définition 53 (Equivalence observationnelle abstraite) *Supposons* $\Gamma;\Lambda \vdash M, M' : \tau$. *$M$ et M' sont observationellement équivalents, ce qu'on écrit* $M \equiv_A^{\Gamma,\Lambda} M'$, *ssi pour tout contexte* $C[.]$ *tel que* $\cdot;\cdot \vdash C[M], C[M'] : \mathbf{B}$ *et pour tout EQA A on a* $[\![C[M]]\!]_A^{\Gamma,\Lambda} = [\![C[M']]\!]_A^{\Gamma,\Lambda}$.
La classe d'équivalence de M par rapport à cette équivalence observationnelle est dénotée par $\widetilde{M}_A^{\Gamma,\Lambda}$, *par extension on dit que le type de cette classe d'équivalence est* τ.

Définition 54 (Valeurs abstraites) *Etant donné les contextes de typage* $\Gamma;\Lambda;\Delta$, *une valeur abstraite* $v_{A,\tau}^{\Gamma;\Lambda;\Delta}$ *de type* τ, *avec* $\tau \neq \sigma \otimes \sigma'$, *par rapport au contexte* $\Gamma;\Lambda;\Delta$ *et à l'EQA* $A = (\mathcal{R}, \mathcal{P})$ *est soit une classe d'équivalence de type* τ *pour* $\equiv_A^{\Gamma,\Lambda}$, *si* $\tau \neq \mathbf{Q}$, *ou bien une paire* (C, b) *formée par une classe d'équivalence C de $\mathcal{R}$ et un booléen b (l'idée étant que si b est vrai alors le bit quantique dénoté est dans $\mathcal{P}$).*

Si $\tau = \sigma' \otimes \sigma''$, *alors* $v_{A,\tau}^{\Gamma;\Lambda;\Delta}$ *est une paire* (v', v'') *formée de valeurs abstraites de types respectifs* σ', σ''.

L'ensemble des valeurs abstraites relatives à un EQA A, un contexte de typage $\Gamma;\Lambda;\Delta$ *et de type* τ *est dénoté par* $\Xi_{A,\tau}^{\Gamma;\Lambda;\Delta}$.

Les valeurs abstraites servent à définir l'interprétation des variables libres. Comme pour un certain contexte de typage $\Gamma;\Lambda;\Delta$ plus d'un type peut être utilisé nous avons besoin de considérer des collections de valeurs abstraites des différents types apparaissants dans $\Gamma;\Lambda;\Delta$: on note $\Xi_{\Gamma;\Lambda;\Delta}$ l'union disjointe des $\Xi_\tau^{\Gamma;\Lambda;\Delta}$ pour chaque τ dans $\Gamma;\Lambda;\Delta$.

Définition 55 (Modèles) *Un modèle sur* $\Gamma;\Lambda;\Delta$ *est un n-uplet* $\mathcal{M}^{\Gamma;\Lambda;\Delta} = \langle A, \mathcal{I}\rangle$, *avec A un EQA, $\mathcal{I}$ une fonction des variables définies dans* $\Gamma;\Lambda;\Delta$ *vers* $\Xi_{\Gamma;\Lambda;\Delta}$.

Définition 56 (Extensions de modèle) *Soit* $\mathcal{M}^{\Gamma;\Lambda;\Delta} = \langle A, \mathcal{I}\rangle$ *un modèle, alors* $\mathcal{M}'$ *dénoté par* $\mathcal{M} \cdot x : v = \langle A', \mathcal{I}'\rangle$, *est le modèle pour lequel* $v \in \Xi_{A',\tau}^{\Gamma;\Lambda;\Delta}$ *est défini comme suit : le contexte de typage de* $\mathcal{M}'$ *est* $\Gamma;\Lambda;\Delta, x : \tau$. *Si le type de x est* $\tau = \sigma \otimes \sigma'$, *alors v est un couple formé de valeurs abstraites* v', v'' *de types respectifs* σ, σ'. *Si x est de type* $\mathbf{Q}$ *: alors si* $v = (C, \mathbf{0})$ *on a*

$A' = (\mathcal{R} \cup C, \mathcal{P}' \cup \{x\})$, sinon si $v = (C, \mathbf{1})$ on a $A' = (\mathcal{R} \cup C, \mathcal{P}')$. Enfin si x est de type $\sigma \neq \mathbf{Q}$, alors : $\mathcal{I}'(y) = \mathcal{I}(y)$ pour tout $x \neq y$ et $\mathcal{I}'(x) = v$.

Définition 57 (Interprétation des termes) *Soit $\mathcal{M}^{\Gamma,\Lambda} = \langle A, \mathcal{I}, \tau \rangle$ un modèle, l'interprétation d'un terme est définie par : $[\![u]\!]_{\mathcal{M}} = \mathcal{I}(u)$; $[\![q_i]\!]_{\mathcal{M}} = (\overline{q_i}^A, b_i^A)$, où b_i^A est $\mathbf{0}$ ssi $q_i \in \mathcal{P}$ avec $A = \langle \mathcal{R}, \mathcal{P} \rangle$; $[\![\langle e, e' \rangle]\!]_{\mathcal{M}} = \langle [\![e]\!]_{\mathcal{M}}, [\![e']\!]_{\mathcal{M}} \rangle$.*

Définition 58 (Satisfaisabilité) *Une assertion C est satisfaisable dans le modèle $\mathcal{M} = \langle A, \mathcal{I} \rangle$, noté $\mathcal{M} \models C$, si on peut dériver ce jugement des règles suivantes :*

- *$\mathcal{M} \models u \leftrightarrow v$ si $(\pi_1([\![u]\!]_{\mathcal{M}}) = \pi_1([\![v]\!]_{\mathcal{M}}))$.*
- *$\mathcal{M} \models \|u$ si $\pi_2([\![u]\!]_{\mathcal{M}})$ est $\mathbf{0}$.*
- *$\mathcal{M} \models \not\|u$ si $\pi_2([\![u]\!]_{\mathcal{M}})$ est $\mathbf{1}$.*
- *$\mathcal{M} \models C \vee C'$ si $\mathcal{M} \models C$ ou $\mathcal{M} \models C'$.*
- *$\mathcal{M} \models C \wedge C'$ si $\mathcal{M} \models C$ et $\mathcal{M} \models C'$.*
- *$\mathcal{M} \models C \implies C'$ si $\mathcal{M} \models C$ implique $\mathcal{M} \models C'$.*
- *$\mathcal{M} \models \forall u : \sigma.C$ si pour toutes les valeurs abstraites $v \in \Xi_{A,\sigma}^{\Gamma;\Lambda;\Delta}$, et $\mathcal{M}' = \mathcal{M} \cdot u : v$, on a $\mathcal{M}' \models C$.*
- *$\mathcal{M} \models \exists u : \sigma.C$ s'il existe une valeur abstraite v telle que si $\mathcal{M}' = \mathcal{M} \cdot u : v$, alors $\mathcal{M}' \models C$.*
- *$\mathcal{M} \models \{C\} e_1 \bullet e_2 = e_3 \{C'\}$ si pour tous les modèles $\mathcal{M}'^{\Gamma;\Lambda;\Delta} = \langle A', \mathcal{I}' \rangle$ tels que $\mathcal{M}'^{\Gamma;\Lambda;\Delta} \models C$, avec les conditions suivantes : $\Gamma; \Lambda; \Delta \vdash e_1 : \sigma \to \tau$, et $\Gamma; \Lambda; \Delta \vdash e_2 : \sigma$ tel que pour tous termes $t_1, t2$ avec $[\![t_i]\!]_{\mathcal{M}'} = [\![e_i]\!]_{\mathcal{M}'}$ pour $i \in \{1, 2\}$ on ait :*
  - *$[A, (t_1 \ \ t_2)] \to_{\mathcal{A}}^* [A', V]$*
  - *avec deux sous-cas : 1) τ est $\mathbf{Q}$ et $V = q_i$ et $\mathcal{M}' = \mathcal{M} \cdot e_3 : (\overline{q_i}^{A'}, q_i \in \mathcal{P}_{A'})$. 2) τ n'est pas $\mathbf{Q}$ et $\mathcal{M}' \cdot e_3 : \widetilde{V}_A^{\Gamma;\Lambda;\Delta,e_3:\tau} \models C'$*

3.3.4 Règles d'inférences de la logique d'enchevêtrement

Nous donnons les règles pour dériver les jugements de la forme $\{C\} M :^{\Gamma;\Lambda;\Delta;\tau} u\{C'\}$. Dans ce jugement u est lié dans C', donc u ne peut pas apparaître librement dans C. u est la contrepartie logique de la valeur calculée par M.

Les règles se divisent en deux groupes : le premier est formé des règles basées sur la structure de M, le second groupe est composé de règles purement logiques.

Définition 59 (Règles structurelles) *Soit $\Gamma; \Lambda \vdash M : \tau$, on définit inductivement le jugement $\{C\} M :^{\Gamma;\Lambda;\Delta;\tau} u\{C'\}$ par les règles données dans la figure 3.7 page 69.*

Avec pour la règle $[HAD_J]$, s'il existe C'' tel que $C'' \wedge \|u \equiv C'$ l'assertion $C'[\neg\|v]$ est $C'' \wedge \neg\|u$ sinon $C'\neg\|u$. Pour la règle $[MEAS_J]$, l'assertion $C'[-u]$

est C' pour lequel toutes les assertions contenant u ont été effacées. Dans $[ABS_J]$, C^{-x} signifie que x n'apparait pas librement dans C. Dans $[VAR_J]$, $C[u/x]$ est l'assertion C dans laquelle toutes le occurrences libres de x ont été remplacées par u.

Les jugements de la partie purement fonctionelle sont standard et nous nous référons à [BHY05]. Nous introduisons juste une manière de gérer les couples de manière à simplifier les manipulations, mais nous aurions pu utiliser les projections plutôt que d'introduire deux nouveaux noms. En ce qui concerne le fragment des actions quantiques, la $[CNOT1_J]$ n'a pas d'influence sur l'enchevêtrement puisque le premier argument du $\mathfrak{Cnot}$ est un état de base. Dans la règle $[CNOT2_J]$, $C' \wedge u \leftrightarrow v$ introduit de l'enchevêtrement entre les deux arguments de l'opérateur $\mathfrak{Cnot}$. On peut noter qu'il n'est pas nécessaire d'introduire la relation d'enchevêtrement entre toutes les paires de bits quantiques que cette opération peut engendrer. En effet, comme la relation d'enchevêtrement est une relation d'équivalence on peut toujours ajouter aux jugements (cf. les règles logiques qui sont définies dans la définition 60, notamment la règle [*promote*]) des formules pour exprimer la transitivité, reflexivité et la symétrie de la relation d'enchevêtrement, par exemple $\forall x, y, z : \mathbf{Q}.x \leftrightarrow y \wedge y \leftrightarrow z \implies x \leftrightarrow z$ pour la transitivité. En effet, tout EQA correct, par définition, va valider ce jugement. Nous les supposerons définis de manière implicite dans la suite du texte. Comme nous l'avons vu l'opérateur de phase ne change pas le fait qu'un état soit de base ou pas, alors que l'opérateur d'Hadamard met, a priori, le bit quantique dans un état qui n'est pas un état de base, ce qui justifie les conclusions des règles $[HAD_J]$ et $[PHASE_J]$.

Définition 60 (Règles logiques)

$$\frac{\{C_0\}V : u\{C_0'\} \quad C \vdash C_0' \quad C_0 \vdash C'}{\{C\}V : u\{C'\}}[LOG_J]$$

$$\frac{\{C\}V : u\{C'\}}{\{C \wedge C_0\}V : u\{C' \wedge C_0\}}[promote]$$

$$\frac{\{C \wedge C_0\}V : u\{C'\}}{\{C\}V : u\{C_0 \implies C'\}}[\implies ELim]$$

$$\frac{\{C\}M : u\{C_0 \implies C'\}}{\{C \wedge C_0\}V : u\{C'\}}[\wedge Elim]$$

$$\frac{\{C_1\}M : u\{C\} \quad \{C_2\}M : u\{C\}}{\{C_1 \vee C_2\}M : u\{C\}}[\vee L]$$

$$\frac{\{C\}M : u\{C_1\} \quad \{C\}M : u\{C_2\}}{\{C\}M : u\{C_1 \wedge C_2\}}[\wedge R]$$

$$\frac{\{C\}M : u\{C'^{-x}\}}{\{\exists x.C\}M : u\{C'\}}[\exists L]$$

$$\frac{\{C^{-x}\}M : u\{C'\}}{\{C\}M : u\{\forall x.C'\}}[\forall R]$$

avec $C \vdash C'$ qui est une preuve standard de la logique du premier ordre (voir par exemple [Smu68]), et où C^{-x} est une assertion dans laquelle x n'apparait pas librement.

Théorème 8 (Correction) *Supposons que $\{C\}M :^{\Gamma;\Lambda;\Delta;\tau} u\{C'\}$ est prouvable. Alors pour tout modèle $\mathcal{M} = \langle A, \mathcal{I}\rangle$, pour tout EQA A', pour toute valeur abstraite v tels que :*

1. $\mathcal{M} \models C$.
2. $[A, M] \rightarrow_{\mathcal{A}}^{*} [A', V]$.
3. $v \in \Xi_{A',\tau}^{\Gamma;\Lambda;\Delta}$.

alors $\mathcal{M} \cdot u : v \models C'$.

Exemple 3 *Ce petit exemple montre comment la logique d'enchevêtrement peut être utilisée pour analyser des propriétés non locales et non compositionnelles. On cherche à prouver le jugement suivant :*

$$\{\mathsf{T}\}P : u\{\forall x, y, z, t.\{x \leftrightarrow y \wedge z \leftrightarrow t\}u \bullet y, z = v\{x \leftrightarrow t\}\}$$

avec P le programme suivant :

$$\lambda y, z : \mathbf{Q}.\mathsf{let}\ \langle u, v\rangle = (\mathfrak{Cnot}\ \langle y, z\rangle)\ \mathsf{in}\ \langle(\mathsf{meas}\ u), (\mathsf{meas}\ v)\rangle$$

En utilisant la règle $[APP_J]$ il est possible de dériver le jugement suivant sur les bits quantiques concrets :

$$\{C\}(P\ \langle q_2, q_3\rangle) : \langle u, v\rangle\{q_1 \leftrightarrow q_4\}$$

où C dénote l'assertion suivante : $q_1 \leftrightarrow q_2 \wedge q_3 \leftrightarrow q_4$. Ce jugement est remarquable en ce qu'il établit des résultats sur l'enchevêtrements des bits quantiques q_1, q_4 alors même que ces deux bits quantiques n'apparaissent pas dans le morceau de code analysé.

Contrairement aux résultats classiques, voir [BHY05], la logique d'enchevêtrement n'est pas complète. En effet, si on considère le programme suivant, qui équivaut à une négation :

$$NOT = \lambda x.(\mathfrak{H}\ (\mathfrak{T}\ (\mathfrak{T}\ (\mathfrak{H}\ x))))$$

alors on voudrait pouvoir inférer un jugement comme celui qui suit :

$$\{\|q_1\}(NOT\ q_1) : u\{\|u\}$$

mais cela n'est pas possible à cause de la règle $[HAD_J]$.

3.3.5 Travaux reliés

Les premiers travaux traitants de l'enchevêtrement, et de son dual la séparabilité, ont proposé l'utlisation d'un système de typage pour approximer la relation d'enchevêtrement sur un tableau de bit quantiques [Per07]. Dans [Per08], l'analyse est faite au moyen d'une interprétation abstraite. La différence notable entre ces deux travaux et la logique que nous avons présenté réside dans le fait que dans [Per08] le langage de programmation considéré est impératif avec une construction `while` : il permet donc l'éxécution de programmes infinis, alors que notre langage de programmation est normalisant (c'est une version du λ-calcul simplement typé). Cependant notre logique permet de prendre en compte l'ordre supérieur, ce que ne permettent pas les travaux pré-cités. Un objectif serait d'ajouter un point fixe au langage. Cela aurait des implications non triviales en termes sémantiques.

Il est à noter que la logique introduite n'est pas très puissante car elle ne peut pas permettre de parler du fait que deux qubits sont séparables et ne contient pas de fragment négatif. Une piste pour remédier à cet état de fait serait de modifier la règle $[CNOT2_J]$ en intégrant une modalité car il faut retirer le fait que $u \not\leftrightarrow v$ de la formule C' (ce qui n'est pas évident à faire étant donné les multiples implications que cela peut avoir pour la formule C'). D'autre part pour intégrer la négation il faut traiter correctement la commutation de la double négation commute l'approximation faite par la sémantique (i.e. $u \leftrightarrow v$ signifiant que u et v sont peut être enchevêtrés mais $u \not\leftrightarrow v$ devrait signifier qu'on est sûr que u n'est pas enchevêtré avec v).

$$\frac{}{\Gamma;\cdot;\kappa \vdash \mathbf{0} : \mathbf{B}}[AxT] \qquad \frac{i \in \kappa}{\Gamma;\Lambda;\kappa \setminus \{i\} \vdash q_i : \mathbf{Q}}[AxQ] \qquad \frac{}{\Gamma;\cdot;\kappa \vdash \mathbf{1} : \mathbf{B}}[AxF]$$

$$\frac{}{\Gamma;x:\sigma;\kappa \vdash x : \sigma}[VarQ] \qquad \frac{\Gamma;\cdot;\kappa \vdash M : \sigma}{\Gamma,x:\tau;\cdot;\kappa \vdash M : \sigma}[Wkg] \qquad \frac{\sigma \text{ intuitionniste}}{\Gamma,x:\sigma;\cdot;\kappa \vdash x : \sigma}[Var]$$

$$\frac{\Gamma;\Lambda;\kappa \vdash M : \tau}{\Gamma;\Lambda,x:\sigma;\kappa \vdash M : \tau}[WkgL]$$

$$\frac{\Gamma,x:\tau,y:\tau;\Lambda;\kappa \vdash M : \sigma}{\Gamma,z:\tau;\Lambda;\kappa \vdash M[x := z; y := z] : \sigma}[CTN]$$

$$\frac{\Gamma;\Lambda_1;\kappa \vdash M : \sigma \to \tau \quad \Gamma;\Lambda_2;\kappa \vdash N : \sigma}{\Gamma;\Lambda_1,\Lambda_2;\kappa \vdash (M \ \ N) : \tau}[\to^\circ E]$$

$$\frac{\Gamma;\Lambda,x:\sigma;\kappa \vdash M : \tau}{\Gamma;\Lambda;\kappa \vdash \lambda x{:}\sigma.M : \sigma \to \tau}[\to^\circ I]$$

$$\frac{\tau \text{ intuitionniste} \quad \Gamma;\cdot;\cdot \vdash M : \tau \quad \Gamma,x:\tau;\Lambda;\kappa \vdash N : \sigma}{\Gamma;\Lambda;\kappa \vdash \mathsf{let}\ x : \tau = M\ \mathsf{in}\ N : \sigma}[\to IE]$$

$$\frac{\Gamma;\Lambda_1;\kappa_1 \vdash M : \tau \quad \Gamma;\Lambda_2;\kappa_2 \vdash N : \sigma}{\Gamma;\Lambda_1,\Lambda_2;\kappa_1,\kappa_2 \vdash \langle M,N\rangle : \tau \otimes \sigma}[\otimes I]$$

$$\frac{\Gamma;\Lambda_1;\kappa_1 \vdash M : \tau_1 \otimes \tau_2 \quad \Gamma;\Lambda_2,x:\tau_1,y:\tau_2;\kappa_2 \vdash N : \sigma}{\Gamma;\Lambda_1,\Lambda_2;\kappa_1,\kappa_2 \vdash \mathsf{let}\ \langle x,y\rangle : \tau_1 \otimes \tau_2 = M\ \mathsf{in}\ N : \sigma}[\otimes E]$$

$$\frac{\Gamma;\Lambda_1 \vdash M : \mathbf{B} \quad \Gamma;\Lambda_2;\kappa_1 \vdash N : \tau \quad \Gamma;\Lambda_2;\kappa_2 \vdash P : \tau}{\Gamma;\Lambda_1,\Lambda_2;\kappa_1,\kappa_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : \tau}[IF\ I]$$

$$\frac{\Gamma;\Lambda;\kappa \vdash M : \mathbf{Q}}{\Gamma;\Lambda;\kappa \vdash \mathfrak{H}\ M : \mathbf{Q}}[HAD] \qquad \frac{\Gamma;\Lambda;\kappa \vdash M : \mathbf{Q}}{\Gamma;\Lambda;\kappa \vdash \mathfrak{T}\ M : \mathbf{Q}}[PHA]$$

$$\frac{\Gamma;\Lambda;\kappa \vdash M : \mathbf{Q}}{\Gamma;\Lambda;\kappa \vdash \mathsf{meas}\ M : \mathbf{B}}[MEAS] \qquad \frac{\Gamma;\Lambda;\kappa \vdash M : \mathbf{Q} \otimes \mathbf{Q}}{\Gamma;\Lambda;\kappa \vdash \mathfrak{Cnot}\ M : \mathbf{Q} \otimes \mathbf{Q}}[CNOT]$$

FIGURE 3.2 – Règles de typages de λ_L^Q

$$\frac{}{[|\varphi\rangle, (\mathfrak{T}\ \ q_i)] \to_1 [\mathfrak{T}^i(|\varphi\rangle), q_i]}[PHS] \qquad \frac{}{[|\varphi\rangle, (\mathfrak{H}\ \ q_i)] \to_1 [\mathfrak{H}^i(|\varphi\rangle), q_i]}[HDR]$$

$$\frac{}{[\alpha\,|\varphi_{\mathbf{1}}\rangle + \beta\,|\varphi_{\mathbf{0}}\rangle, (\mathsf{meas}\ \ q_i)] \to_{|\alpha|^2} [|\varphi_{\mathbf{1}}\rangle, \mathbf{0}]}[MEF]$$

$$\frac{}{[\alpha\,|\varphi_{\mathbf{1}}\rangle + \beta\,|\varphi_{\mathbf{0}}\rangle, (\mathsf{meas}\ \ q_i)] \to_{|\beta|^2} [|\varphi_{\mathbf{0}}\rangle, \mathbf{1}]}[MET]$$

$$\frac{}{[|\varphi\rangle, (\mathfrak{Cnot}\ \ \langle q_i, q_j\rangle)] \to_1 [\mathfrak{Cnot}^{i,j}(|\varphi\rangle), \langle q_i, q_j\rangle]}[CNO]$$

FIGURE 3.3 – Sémantique opérationnelle de λ_L^Q, fragment quantique

$$\frac{(u:\tau) \in \Gamma; \Lambda\Delta}{\Gamma; \Lambda; \Delta \vdash u : \tau}[TAsAx]$$

$$\frac{}{\Gamma; \Lambda; \Delta \vdash \mathbf{c} : \mathbf{B}}[TCons_{\mathbf{c}}] \qquad \frac{}{\Gamma; \Lambda; \Delta \vdash q_i : \mathbf{Q}}[TAsQ]$$

$$\frac{\Gamma; \Lambda; \Delta \vdash e : \tau \quad \Gamma; \Lambda; \Delta \vdash e' : \tau'}{\Gamma; \Lambda; \Delta \vdash \langle e, e'\rangle : \tau \otimes \tau'}[TAs\otimes] \qquad \frac{\Gamma; \Lambda; \Delta \vdash u : \tau_1 \otimes \tau_2}{\Gamma; \Lambda; \Delta \vdash \pi_i(u) : \tau_i}[TAs\pi_i]$$

FIGURE 3.4 – Typage des termes logiques

$$\frac{\Gamma; \Lambda; \Delta \vdash e : \mathbf{Q}}{\Gamma; \Lambda; \Delta \vdash \|e}[TAs\|] \qquad \frac{\Gamma; \Lambda; \Delta \vdash e : \tau \quad \Gamma; \Lambda; \Delta \vdash e' : \tau}{\Gamma; \Lambda; \Delta \vdash e = e'}[TAs =]$$

$$\frac{\Gamma; \Lambda; \Delta \vdash C \quad \Gamma; \Lambda; \Delta \vdash C'}{\Gamma; \Lambda; \Delta \vdash C \wedge C'}[TAs\wedge] \qquad \frac{\Gamma; \Lambda; \Delta \vdash C}{\Gamma; (\Lambda; \Delta) \setminus u : \sigma \vdash \exists u : \sigma.C}[TAs\exists]$$

$$\frac{\Gamma; \Lambda; \Delta \vdash C \qquad \Gamma; \Lambda; \Delta \vdash e2 : \sigma \quad \Gamma; \Lambda; \Delta, e3 : \tau \vdash C' \quad \Gamma; \Lambda; \Delta \vdash e1 : \sigma \to \tau}{\Gamma; \Lambda, \Lambda; \Delta \vdash \{C\}e1 \bullet e2 = e3\{C'\}}[TAsEV]$$

FIGURE 3.5 – Règles de typage des assertions

$$\frac{}{[(\mathcal{R},\mathcal{P}),(\mathfrak{T}\ \ q_i)] \rightarrow_{\mathcal{A}} [(\mathcal{R},\mathcal{P}),q_i]}[PHS_{\mathcal{A}}]$$

$$\frac{}{[(\mathcal{R},\mathcal{P}),(\mathfrak{H}\ \ q_i)] \rightarrow_{\mathcal{A}} [(\mathcal{R},\mathcal{P}\setminus\{q_i\}),q_i]}[HDR_{\mathcal{A}}]$$

$$\frac{}{[(\mathcal{R},\mathcal{P}),(\mathsf{meas}\ \ q_i)] \rightarrow_{\mathcal{A}} [(\mathcal{R}\setminus q_i,\mathcal{P}\cup\{q_i\}),\mathbf{\frac{0}{1}}]}[MET_{\mathcal{A}}]$$

$$\frac{}{[(\mathcal{R},\mathcal{P}),(\mathfrak{Cnot}\ \ \langle q_i,q_j\rangle)] \rightarrow_{\mathcal{A}} [(\mathcal{R},\mathcal{P}),\langle q_i,q_j\rangle]}[CNO1_{\mathcal{A}}]\text{si } q_i \in \mathcal{P}$$

$$\frac{}{[(\mathcal{R},\mathcal{P}),(\mathfrak{Cnot}\ \ \langle q_i,q_j\rangle)] \rightarrow_{\mathcal{A}} [(\mathcal{R}\cdot q_i \leftrightarrow q_j,\mathcal{P}\setminus\{q_i,q_j\}),\langle q_i,q_j\rangle]}[CNO0_{\mathcal{A}}]\text{si } q_i \notin \mathcal{P}$$

FIGURE 3.6 – Sémantique opérationnelle abstraite, actions quantiques

$$\frac{\{C \wedge \|u\}N :^{\Gamma;\Lambda;\Delta;\mathbf{Q}\otimes\mathbf{Q}} \langle u, v\rangle\{C'\}}{\{C \wedge \|u\}(\mathfrak{Cnot}\ N) :^{\Gamma;\Lambda;\Delta;\mathbf{Q}\otimes\mathbf{Q}} \langle u, v\rangle\{C'\}}[CNOT1_J]$$

$$\frac{\{C\}N :^{\Gamma;\Lambda;\Delta;\mathbf{Q}\otimes\mathbf{Q}} \langle u, v\rangle\{C'\}}{\{C\}(\mathfrak{Cnot}\ N) :^{\Gamma;\Lambda;\Delta;\mathbf{Q}\otimes\mathbf{Q}} \langle u, v\rangle\{C' \wedge u \leftrightarrow v\}}[CNOT2_J]$$

$$\frac{\{C\}N :^{\Gamma;\Lambda;\Delta;\mathbf{Q}} v\{C'\}}{\{C\}(\mathfrak{H}\ N) :^{\Gamma;\Lambda;\Delta;\mathbf{Q}} v\{C'[\not\|v]\}}[HAD_J] \qquad \frac{\{C\}N :^{\Gamma;\Lambda;\Delta;\mathbf{Q}} u\{C'\}}{\{C\}(\mathfrak{T}\ N) :^{\Gamma;\Lambda;\Delta,\mathbf{Q}} u\{C'\}}[PHASE_J]$$

$$\frac{}{\{C[u/x]\}x :^{\Gamma;\Lambda;\Delta,u:\tau;\tau} u\{C\}}[VAR_J] \qquad \frac{c \in \{\mathbf{0}, \mathbf{1}\}}{\{C[u/c]\}c :^{\Gamma;\Lambda;\Delta,u:\mathbf{B};\mathbf{B}} u\{C\}}[CONST_J]$$

$$\frac{\{C\}M :^{\Gamma;\Lambda;\Delta;\mathbf{Q}} u\{C'\}}{\{C\}\mathsf{meas}\ M :^{\Gamma;\Lambda;\Delta,v:\mathbf{B};\mathbf{B}} v\{C'[-u] \wedge \|u\}}[MEAS_J]$$

$$\frac{\{C\}M :^{\Gamma;\Lambda;\Delta;\mathbf{B}} b\{C_0\} \quad \{C_0[\mathbf{0}/b]\}N :^{\Gamma;\Lambda;\Delta;\tau} x\{C'\} \quad \{C_0[\mathbf{1}/b]\}P :^{\Gamma;\Lambda;\Delta;\tau} x\{C'\}}{\{C\}\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P :^{\Gamma;\Lambda;\Delta,u:\tau;\tau} u\{C'\}}[IF_J]$$

$$\frac{\{C\}M :^{\Gamma;\Lambda;\Delta;\sigma\to\tau} m\{C_0\} \quad \{C_0\}N :^{\Gamma;\Lambda;\Delta;\sigma} n\{C_1 \wedge \{C_1\}m \bullet n = u\{C'\}\}}{\{C\}(M\ \ N) :^{\Gamma;\Lambda;\Delta,u:\tau;\tau} u\{C'\}}[APP_J]$$

$$\frac{\{C^{-x} \wedge C_0\}M :^{\Gamma;\Lambda;\Delta;\tau} m\{C'\}}{\{C\}\lambda x{:}\sigma.M :^{\Gamma[-x];\Lambda[-x];\Delta,u:\sigma\to\tau;\sigma\to\tau} u\{\forall x.\{C_0\}u \bullet x = m\{C'\}\}}[ABS_J]$$

$$\frac{\{C\}M :^{\Gamma;\Lambda;\Delta;\tau} m\{C_0\} \quad \{C_0\}N :^{\Gamma;\Lambda;\Delta;\sigma} n\{C'[m/u, n/v]\}}{\{C\}\langle M, N\rangle :^{\Gamma;\Lambda;\Delta,u:\tau,v:\sigma;\tau\otimes\sigma} \langle u, v\rangle\{C']\}}[\times_J]$$

$$\frac{\{C\}M :^{\Gamma;\Lambda;\Delta;\tau_1\otimes\tau_2} m\{C'[\pi_i(m)/u]\} \quad i \in \{1, 2\}}{\{C\}(\pi_i\ M) :^{\Gamma;\Lambda;\Delta u:\tau_i;\tau_i} u\{C'\}}[\pi_J]i$$

FIGURE 3.7 – Règles structurelles

Chapitre 4

Réécriture de graphes et mémoire

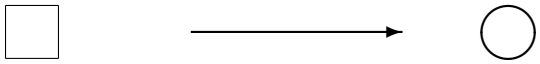
4.1 Introduction

L'utilisation de pointeurs en programmation permet des implantations efficaces. Cette efficacité a un prix : les programmes manipulant des pointeurs sont plus durs à analyser et bien plus sujets aux erreurs. L'analyse des dépendances dans un tel contexte est elle aussi considérablement plus complexe.

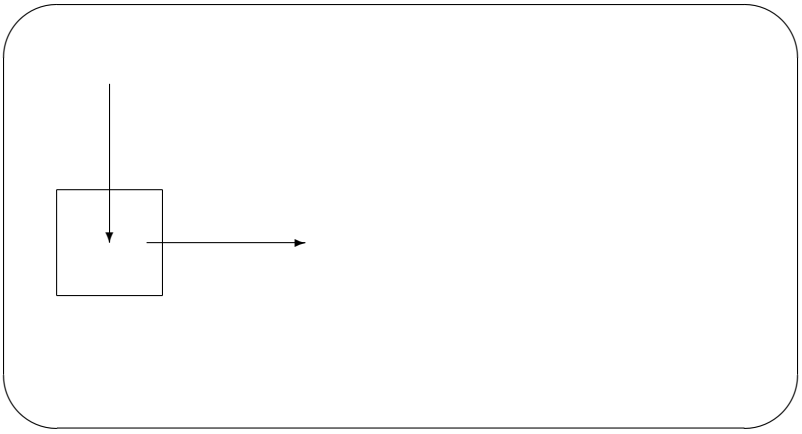
Une approche formelle pour écrire et valider des programmes utilisant des pointeurs est la transformation/réécriture de graphes : on peut voir les graphes comme une modélisation mathématique de la mémoire de l'ordinateur dans laquelle les noeuds représentent les cases mémoires et les arrêtes les pointeurs. Nous nous intéressons au cas des terme-graphes, où les noeuds peuvent être étiquetés (par des symboles de fonction) avec une arité fixée. Ces travaux ont été menés en commun avec Dominique Duval et Rachid Echahed.

Contrairement à la réécriture de termes [BN98], qu'on peut présenter comme un cas particulier de réécriture de graphes où l'on ne considère que des arbres, il n'y a pas vraiment de formalisme canonique pour la réécriture de graphes. En effet l'idéé de la réécriture, simple au premier abord, d'identifer une instance d'une partie gauche pour la remplacer par une instance de la partie droite est simple à réaliser quand il n'y a qu'un unique point d'entrée (la racine de l'arbre dans le cas de la réécriture de termes). Mais quand on considère un graphe en général il faut savoir précisément comment gérer les arrêtes entrantes et sortantes ce qui est loin d'être un problème trivial. Schématiquement on peut représenter le problème de la façon suivant. Etant

donné la règle :



Alors son application au graphe :



n'est pas aussi simple que de retirer le carré pour le remplacer par le cercle comme le demande la règle de réécriture : en effet comment traiter proprement les pointeurs entrants et sortants ? Que devient la cible du pointeur entrant (problème classique des "dangling pointers") ? Que devient la source du pointeur sortant (ce pointeur a-t-il encore un sens) ?

Il y a plusieurs écoles pour traiter les problèmes soulevés par les transformations de graphes. L'approche algorithmique, telle que [BvEG+87], défini en détail chaque étape de la transformation d'un graphe en fournissant un algorithme. Cependant cette approche est très proche d'une implantation et ne permet pas une approche de haut niveau, abstraite, permettant une approche formelle. Dans [AK96], une définition équationnelle de termes-graphes est exploitée pour définir des transformations de termes-graphes. Cependant les transformations obtenues le sont modulo bisimilarité (deux terme-graphes sont bisimilaires s'ils représentent le même terme rationel). Le problème est que cette bisimilarité n'est pas une congruence en général, par exemple les

longueurs de deux listes circulaires différentes mais bisimilaires ne sont pas égales. Une troisième approche, plus abstraite, est l'approche algébrique proposée par Ehrig dans [EPS73]. L'idée est d'utiliser des propriétés générales des catégories (notamment la notion de pushout) pour gérer les détails des transformations de graphes.

C'est cette dernière approche que nous allons utiliser car elle est déclarative et permet d'aborder les transformations de graphes d'un point de vue plus formel et donc d'analyser ces transformations de manière plus abstraite. Nous allons traiter deux types d'analyses liées à la transformation de graphes. Ces deux analyses butent traditionnellement sur une difficulté commune aux systèmes de réécriture de graphes basés sur les pushouts : la difficulté d'effacer une information.

Dans les systèmes basés sur le double pushout les règles de réécriture sont des empans $L \leftarrow K \rightarrow R$ où L, K et R sont des graphes et les flèches représentent des homomorphismes de graphes. Si nous considérons la catégorie des graphes définies dans [DEP06] (cette catégorie est similaire à **Gr** définie en section 4.2 en oubliant le fait qu'il y ait des racines).

L'application d'une telle règle sur un graphe G consiste à trouver un homomorphisme (un filtrage) $m : L \rightarrow G$ et à calculer le graphe H de telle manière à ce que le diagramme suivant :

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle m'} \\ G & \xleftarrow{l'} & D & \xrightarrow{r'} & H \end{array}$$

constitue un double pushout (des conditions sont requises pour assurer l'existence de ce double pushout [EPS73]).

Un des problèmes lié à ce type de définition est le suivant. Considérons la traduction dans ce formalisme de la simple règle de réécriture de termes suivants : $f(x) \rightarrow f(b)$. Quand on applique cette règle au terme $f(a)$, le résultat est le terme $f(b)$. Cependant dans une approche de type double pushout (où les termes sont vus comme des graphes), le terme obtenu n'est pas simplement $f(b)$: il contient également la constante a. En effet, la règle de réécriture de terme $f(x) \rightarrow f(b)$ est traduite par l'empan $f(x) \leftarrow K_0 \rightarrow f(b)$, (le contenu précis de K_0 n'est pas important pour notre propos et dépend du formalisme choisi). Quand on applique cette règle au graphe $f(a)$, en suivant l'approche double-pushout, on obtient le diagramme :

$$\begin{array}{ccccc} f(x) & \longleftarrow & K_0 & \longrightarrow & f(b) \\ \downarrow & & \downarrow & & \downarrow \\ f(a) & \longleftarrow & D_0 & \longrightarrow & H_0 \end{array}$$

où H_0 contient à la fois $f(b)$ et a (à cause des propriétés même des pushouts). En fait a doit appartenir à D_0, car le carré de gauche est un pushout ; d'où, a doit aussi appartenir à H_0, car le carré de droite est un pushout.

Dans la section 4.3 nous nous intéressons au traitement de ces noeuds, dont nous venons de montrer sur un exemple, qu'ils peuvent, après un pas de réécriture, n'êtres plus accessibles (ce que l'on appelle "garbage collection", ou problème de ramassage des miettes) dans un tel cadre. C'est un mécanisme fondamental pour les implantations pour éviter les fuites de mémoire au cours de l'éxécution d'un programme. Nous allons montrer comment on peut, à l'aide d'outils catégoriques, spécifier ce problème de ramasse-miettes d'un point de vue abstrait et mathématique. Cela permet de donner une définition algébrique d'une réécriture "propre", c'est-à-dire dans laquelle il n'y a pas de noeuds non atteignables après un pas de réécriture.

Dans la section 4.4 nous donnons un formalisme de réécriture de graphe différent, basé une simple pushout hétérogène. Cette définition permet au programmeur de gérer directement la mémoire au niveau des règles (en permettant le clonage de noeuds, en prenant en compte le fait que supprimer revient à cloner 0 fois), et nous montrons comment ce formalisme permet de définir une analyse du coût en mémoire d'un programme de transformations de graphes. Cette analyse est simple car il suffit d'étudier ce qui se passe règle par règle, c'est l'avantage de la déclarativité du formalisme. Cette analyse est aussi assez puissante car elle permet de donner des résultats plus forts que d'autres propositions, basées notamment sur le typage (voir [HJ03]).

Nous commençons en section 4.2 par présenter notre formalisme de réécriture de graphe basé sur une approche de type double pushout. Ce formalisme permet notamment des manipulations de pointeurs (des redirections) locales (on peut rediriger une arrête précise) et globales (il est possible de rediriger toutes les arrêtes incidentes à un sommet vers un nouveau sommet).

4.2 Réécriture de graphes enracinés par double pushout

4.2.1 Graphes enracinés

Les graphes enracinés sont définis comme des graphes de termes [BvEG$^+$87], avec un sous-ensemble de noeud distingué appelés les racines. Ces graphes modélisent des structures de données usuelles comportant des pointeurs (comme les listes circulaires). L'addition de racines permet de prendre en compte le fait qu'au cours de l'éxécution d'un algorithme, à cause des redirections de pointeurs, des données peuvent devenir inaccessibles.

Définition 61 (Signature) *Une* signature Ω *est un ensemble de symboles tels que pour chaque symbole* f *dans* Ω *est assigné un entier,* $\mathrm{ar}(f)$*, son* arité*.*

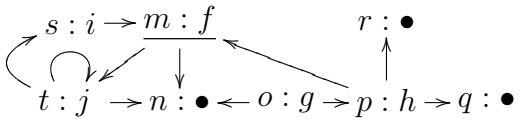
Pour tout ensemble A, l'ensemble des mots sur A est noté A^*, et pour chaque fonction $f : A \to B$, la fonction $f^* : A^* \to B^*$ est l'extension de f sur les mots définies par $f^*(a_1 \dots a_n) = f(a_1) \dots f(a_n)$.

Définition 62 (Graphe) *Un* graphe enraciné G *est un quintuplet* $(\mathcal{N}_G, \mathcal{N}_G^R, \mathcal{N}_G^\Omega, \mathcal{L}_G, \mathcal{S}_G)$ *où* $\mathcal{N}_G$ *est l'ensemble des* noeuds *de* G*,* $\mathcal{N}_G^R \subseteq \mathcal{N}_G$ *est l'ensemble des* racines*,* $\mathcal{N}_G^\Omega \subseteq \mathcal{N}_G$ *est l'ensemble des* noeuds étiquetés*,* $\mathcal{L}_G : \mathcal{N}_G^\Omega \to \Omega$ *est la* fonction d'étiquetage*, and* $\mathcal{S}_G : \mathcal{N}_G^\Omega \to \mathcal{N}_G^*$ *est la* fonction successeur *telle que, pour chaque noeud étiqueté* n*, la taille du mot* $\mathcal{S}_G(n)$ *est l'arité de l'opération* $\mathcal{L}_G(n)$*.*

L'arité d'un noeud n est l'arité de son label et le i^{eme} successeur d'un noeud n est noté $\mathrm{succ}_G(n, i)$. Les *arrêtes* d'un graphe G sont des paires (n, i) où $n \in \mathcal{N}_G^\Omega$ et $i \in \{1, \dots, \mathrm{ar}(n)\}$, le *but* de cette arrête est le noeud $\mathrm{tgt}(n, i) = \mathrm{succ}_G(n, i)$. L'ensemble des arrêtes de G est dénoté par $\mathcal{E}_G$. Le fait que $f = \mathcal{L}_G(n)$ est écrit $n : f$; un noeud non étiqueté n de G est écrit $n : \bullet$ (on peut le voir comme une variable anonyme : c'est l'adresse du noeud qui donne son identité). L'ensemble des noeuds non étiquetés est dénoté par $\mathcal{N}_G^\mathcal{X}$, et on a $\mathcal{N}_G = \mathcal{N}_G^\Omega + \mathcal{N}_G^\mathcal{X}$, où "+" est l'union disjointe.

Exemple 4 *Soit* G *le graphe défini par* $\mathcal{N}_G = \{m, n, o, p, q, r, s, t\}$*,* $\mathcal{N}_G^\Omega = \{m, o, p, s, t\}$*,* $\mathcal{N}_G^\mathcal{X} = \{n, q, r\}$*,* $\mathcal{L}_G$ *est défini par :* $[m \mapsto f, o \mapsto g, p \mapsto h, s \mapsto i, t \mapsto j]$*,* $\mathcal{S}_G$ *est défini par :* $[m \mapsto no, o \mapsto np, p \mapsto qrm, s \mapsto m, t \mapsto tsn]$*, et les racines de* G *sont* $\mathcal{N}_G^R = \{m\}$*.*

G est graphiquement représenté par :

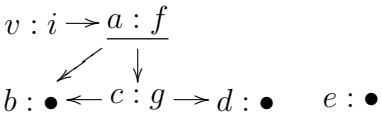


Les racines sont soulignées. Généralement dans nos exemples l'ordre des successeurs est clair à partir du contexte ou sans importance particulière.

Définition 63 (Homomorphisme de graphe) *Un* homomorphisme de graphe enraciné $\varphi : G \to H$ *est une fonction* $\varphi : \mathcal{N}_G \to \mathcal{N}_H$ *conservant les racines, les noeuds étiquetés, la fonction d'étiquetage et les fonctions successeurs. C'est à dire,* $\varphi(\mathcal{N}_G^R) \subseteq \mathcal{N}_H^R$*,* $\varphi(\mathcal{N}_G^\Omega) \subseteq \mathcal{N}_H^\Omega$*, et pour chaque noeud étiqueté* n*,* $\mathcal{L}_H(\varphi(n)) = \mathcal{L}_G(n)$ *et* $\mathcal{S}_H(\varphi(n)) = \varphi^*(\mathcal{S}_G(n))$*.*

L'image $\varphi(n, i)$ *d'une arrête* (n, i) *de* G *est définie comme l'arrête* $(\varphi(n), i)$ *de* H*.*

Exemple 5 *Soit H le graphe suivant :*



Soit $\varphi : \mathcal{N}_H \to \mathcal{N}_G$, où G est le graphe G de l'exemple 4, défini par : $\varphi = [a \mapsto m, b \mapsto n, c \mapsto o, d \mapsto p, e \mapsto p, v \mapsto s]$. Alors, φ est un homomorphisme de H vers G.

4.2.2 Déconnections

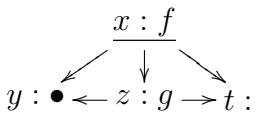
La déconnection d'un graphe L est composée d'un graphe K et d'un homomorphisme de graphe $l : K \to L$. Intuitivement, K est obtenu en redirigeant certaines arrêtes de L vers de nouveaux buts non étiquetés, et l'homomorphisme l reconnecte toutes les arrêtes déconnectées : $\mathcal{N}_K$ contient $\mathcal{N}_L$ plus de nouveaux noeuds non étiquetéss et l est l'identité sur $\mathcal{N}_L$.

Définition 64 (Kit de déconnection) *Un* kit de déconnection $k = (E_l, N_g, E_g)$ *pour un graphe L, où E_l, E_g sont des sous-ensembles de $\mathcal{E}_L$ et $N_g \subseteq \mathcal{N}_L$ contient un ensemble d'arrêtes E_l, qui sont les* arrêtes localement redirigées*, un ensemble de noeuds N_g, appelé les* noeuds globalement redirigés*, et un autre ensemble d'arrêtes E_g, qui sont* les arrêtes globalement redirigées *qui sont distinctes des arrêtes de E_l et telles que le but de chaque arrête de E_g est dans N_g. Un kit de déconnection $(E_l, N_g, \emptyset)$ est simplement écrit (E_l, N_g).*

Définition 65 (Déconnection d'un graphe) *Soit L un graphe et un kit de déconnection $k = (E_l, N_g, E_g)$. Soit K le graphe défini par : $\mathcal{N}_K = \mathcal{N}_L + \mathcal{N}_E + \mathcal{N}_N$, où $\mathcal{N}_E$ est constitué d'un nouveau noeud $n[i]$ pour chaque arrête $(n, i) \in E_l$, $\mathcal{N}_N$ est constitué d'un nouveau noeud $n[0]$ pour chaque noeud $n \in N_g$, $\mathcal{N}_K^R$ contient un noeud pour chaque racine n de L : le noeud n lui même si $n \notin N_g$, $n[0]$ si $n \in N_g$. $\mathcal{N}_K^\Omega = \mathcal{N}_L^\Omega$, pour chaque $n \in \mathcal{N}_L^\Omega$: $\mathcal{L}_K(n) = \mathcal{L}_L(n)$, pour chaque $n \in \mathcal{N}_L^\Omega$ et $i \in \{1, \ldots, \mathrm{ar}(n)\}$: si $(n, i) \notin E_l + E_g$ alors $\mathrm{succ}_K(n, i) = \mathrm{succ}_L(n, i)$, si $(n, i) \in E_l$ alors $\mathrm{succ}_K(n, i) = n[i]$, si $(n, i) \in E_g$ alors $\mathrm{succ}_K(n, i) = \mathrm{tgt}(n, i)[0]$.*
Soit $l : \mathcal{N}_K \to \mathcal{N}_L$ la fonction définie par : $l(n) = n$ si $n \in \mathcal{N}_L$, $l(n[i]) = \mathrm{succ}_L(n, i)$ si $(n, i) \in E_l$, $l(n[0]) = n$ si $n \in N_g$.

Il est clair que l préserve les racines, les noeuds étiquetés et les fonctions d'étiquetage et successeur, donc c'est un homomorphisme de graphe. Alors $l : K \to L$ est la déconnection de L vis-à-vis de k.

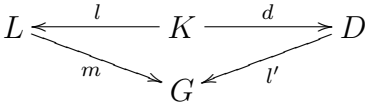
Exemple 6 *Soit L_6 le graphe suivant :*



Considérons les kits de déconnection suivant $k_1 = (\{(x,1),(z,2)\},\emptyset)$ et $k_2(\{(x,3)\},\{x\})$ nous avons les déconnections respectives de L_6, K_6 et K'_6 :

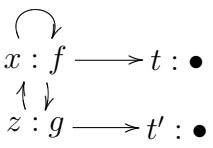
Définition 66 (Filtrage) *Soit L un graphe avec un kit de déconnection $k = (E_l, N_g)$. Un* filtrage de L cohérent avec k *est un homomorphisme de graphe $m : L \to G$ tel que la restriction de m à $(\mathcal{N}_L^\Omega \cup N_g)$ est injective.*

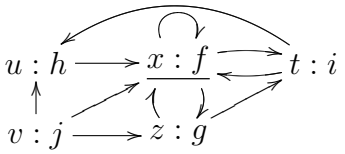
Définition 67 (Déconnection d'un filtrage) *Soit L un graphe, muni d'un kit de déconnection $k = (E_l, N_g)$, et soit $m : L \to G$ un filtrage de L cohérent avec k. Soit $E'_l = m(E_l)$ et $N'_g = m(N_g)$ (comme m est un filtrage, les restrictions de m sont des bijections : $E_l \cong E'_l$ et $N_g \cong N'_g$). Soit E'_g l'ensemble des arrêtes de $G - m(L)$ dont le but est dans N'_g, et soit $k' = (E'_l, N'_g, E'_g)$. Soit $l : K \to L$ la déconnection de L vis-à-vis de k, et $l' : D \to G$ la déconnection de G vis-à-vis k'. Soit $d : \mathcal{N}_K \to \mathcal{N}_D$ une fonction définie par : $d(n) = m(n)$ si $n \in \mathcal{N}_L$, $d(n[i]) = m(n)[i]$ si $n[i] \in \mathcal{N}_E$, $d(n[0]) = m(n)[0]$ si $n[0] \in \mathcal{N}_N$.*
Il est clair que d est un morphisme de graphe. Le diagramme suivant dans **Gr** *est appelé* la déconnection de m vis-à-vis de k *:*



En d'autres mots la déconnection d'un filtrage vis-à-vis d'un kit de déconnection consiste dans la construction de, D (et des morphismes appropriés) étant donnés K, L, G et m, l. D peut être informellement décrit comme étant formé de trois parties. La première est constituée de la partie de G qui n'est pas filtrée. La seconde est l'image de L dans G, enfin il y a les noeuds non étiquetés qui sont introduits pour implanter les redirections ($m(n)[i]$ pour les redirections locales, $m(n)[0]$ pour les globales).

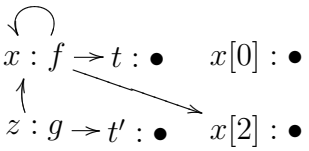
Exemple 7 *Considérons les graphes suivants L_7, G_7 :*

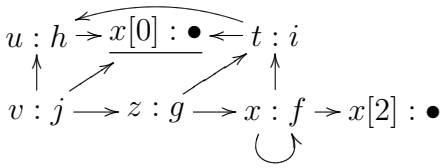
$L_7 =$ 

$G_7 =$ 

et soit k le kit de déconnection $(\{(x,2)\},\{x\})$. L'arrête $(x,2)$ étant l'arrête joignant le noeud x au noeud z.

l'homomorphisme de graphe $m_7 = [x \mapsto x, t \mapsto t, t' \mapsto t, z \mapsto z]$, est un morphisme de L_7 vers G_7. C'est aussi un filtrage de L_7 cohérent avec k. Par déconnection de G_7 vis-à-vis de k nous avons les graphes K_7 qui est la déconnection L_7 vis-à-vis du kit de déconnection $(\{(x,2)\},\{x\})$, et D_7 qui est la déconnection de G_7 vis-à-vis du kit de déconnection $(\{(x,2)\},\{x\},\{(u,1),(v,2),(t,2)\})$:

$K_7 =$ 

$D_7 =$ 

4.2.3 Réécriture de graphes enracinés

Définition 68 (Règle de réécriture) *Une* règle de réécriture*, ou encore une* production*, est un empan de graphes $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ où l est la déconnection de L vis-à-vis du kit de déconnection $k = (E_l, N_g)$, et la restriction de r sur $\mathcal{N}_L^{\mathcal{X}}$ est injective et a ses valeurs dans $\mathcal{N}_R^{\mathcal{X}}$. Alors p est une règle de réécriture* cohérente avec k.

Un pas de réécriture est défini à partir d'un règle de réécriture $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ et d'un filtrage $m : L \to G$, et par rapport à un kit de déconnection $k = (E_l, N_g)$ de L. Le rôle du pas de réécriture consiste en :

1. Ajouter à G une instance de la partie droite R de p,
2. effectuer des redirections locales d'arrêtes de G : chaque arrête (n,i) dans $m(E_l)$ est redirigée vers son nouveau but $n[i]$,
3. effectuer des redirections globles d'arrêtes dans G : toutes les arrêtes incidentes d'un noeud n dans $m(N_g)$, exceptées les arrêtes qui sont des filtrées, sont redirigées vers le nouveau but $n[0]$,
4. modifier les racines de G : si n est une racine de G qui n'est pas dans $m(N_g)$ alors elle reste une racine, mais si n est une racine de G qui est dans $m(N_g)$ alors $n[0]$ devient une racine à la place de n.

Lemme 6 *Soit $\Phi : \mathbf{A} \to \mathbf{A}'$ un foncteur fidèle. Soit $\Sigma = (A_1 \overset{f_1}{\leftarrow} A_0 \overset{f_2}{\rightarrow} A_2)$ un empan et Γ un diagramme dans $\mathbf{A}$:*

$$\begin{array}{ccccc} A_1 & \xleftarrow{f_1} & A_0 & \xrightarrow{f_2} & A_2 \\ & {\scriptstyle g_1}\searrow & & \swarrow{\scriptstyle g_2} & \\ & & A & & \end{array}$$

Si $\Phi(\Gamma)$ est un pushout dans $\mathbf{A}'$ et si pour tout co-cône Δ sur Σ dans $\mathbf{A}$, il y a un morphisme $h : A \to B$ dans $\mathbf{A}$ (B étant un sommet de Δ) tel que $\Phi(h)$ est la co-factorisation de $\Phi(\Delta)$ vis-à-vis de $\Phi(\Gamma)$, alors Γ est un pushout dans $\mathbf{A}$.

Pour chaque empan $\Sigma = (N_1 \overset{\varphi_1}{\leftarrow} N_0 \overset{\varphi_2}{\rightarrow} N_2)$, on dénote par $\sim$ la relation d'équivalence induite par Σ sur $N_1 + N_2$, ce qui signifie qu'elle est engendrée par $\varphi_1(n_0) \sim \varphi_2(n_0)$ pour tout $n_0 \in N_0$, et soit N le quotient $N = (N_1 + N_2)/\sim$. Pour $i \in \{1, 2\}$, soit $\psi_i : N_i \to N$ la fonction qui associe chaque noeud n_i de G_i à sa classe modulo $\sim$. Alors, il est bien connu que le diagramme suivant est un pushout dans **Set**, qui sera appelé *canonique* :

$$\begin{array}{ccccc} & & N_0 & & \\ & {\scriptstyle \varphi_1}\swarrow & & \searrow{\scriptstyle \varphi_2} & \\ N_1 & & & & N_2 \\ & {\scriptstyle \psi_1}\searrow & & \swarrow{\scriptstyle \psi_2} & \\ & & N & & \end{array}$$

Définition 69 (Empan de graphe fortement étiqueté) *Un empan $\Sigma = (G_1 \overset{\varphi_1}{\leftarrow} G_0 \overset{\varphi_2}{\rightarrow} G_2)$ dans* **Gr** *est* fortement étiqueté *si, à partir du moment où deux noeuds étiquetés de $\mathcal{N}(G_1) + \mathcal{N}(G_2)$ sont équivalents par rapport à Σ, alors ils ont la même étiquette et des successeurs équivalents.*

Définition 70 (Carré canonique de graphes) *Soit $\Sigma = (G_1 \overset{f_1}{\leftarrow} G_0 \overset{f_2}{\rightarrow} G_2)$ un empan fortement étiqueté de* **Gr**. *Le* carré canonique de Σ *est le carré suivant dans* **Gr** *:*

$$\begin{array}{ccccc} G_1 & \xleftarrow{\varphi_1} & G_0 & \xrightarrow{\varphi_2} & G_2 \\ & {\scriptstyle \psi_1}\searrow & & \swarrow{\scriptstyle \psi_2} & \\ & & G & & \end{array}$$

dans lequel le carré sous-jascent sur les noeuds est le pushout canonique dans **Set**, *un noeud n dans G est une racine si et seulement si $n = \psi_i(n_i)$ pour une racine n_i dans G_1 ou G_2, un noeud n dans G est étiqueté si et seulement si $n = \psi_i(n_i)$ pour un noeud étiqueté n_i dans G_1 ou G_2, et de plus l'étiquette de n est l'étiquette de n_i et les successeurs de n sont les classes d'équivalences des des successeurs de n_i.*

Il est clair que $\Gamma(\Sigma)$ est un carré commutatif dans **Gr**, c'est même un pushout dans **Gr**.

Lemme 7 (Pushout de graphes non enracinés) *Soit Σ un empan fortement étiqueté de* **Gr**, *et soit Γ le carré canonique sur Σ. Alors $U_0(\Gamma)$ est un pushout dans* $\mathbf{Gr}_0$.

Théorème 9 (Pushout de graphes enracinés) *Soit Σ un empan fortement étiqueté de* **Gr**, *et soit Γ le carré canonique sur Σ. Alors Γ est un pushout dans* **Gr**.

Théorème 10 (Pushout complément) *Soit L un graphe muni d'un kit de déconnection k et soit m le filtrage de L cohérent avec k. Le carré de déconnection de m vis-à-vis de k est un pushout dans la catégorie des graphes.*

Le théorème 10 signifie que d et l' forment un *pushout complément* de l et m. D'autres pushouts compléments de l et m peuvent être obtenus en replaçant E'_g, dans la définition 67, par un de ses sous-ensembles.

Théorème 11 (Pushout direct) *Soit $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ une règle de réécriture et $m : L \rightarrow G$ un filtrage, les deux étants cohérents avec le kit de déconnection k de L. Alors l'empan $D \stackrel{d}{\leftarrow} K \stackrel{r}{\rightarrow} R$ est fortement étiqueté, de telle manière que le carré canonique sur ce empan est un pushout.*

Définition 71 (Pas de réécriture) *Soit $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ une règle de réécriture et $m : L \rightarrow G$ un filtrage, les deux cohérents avec un kit de déconnection k de L. Alors G* se réécrit en H par la régle p *s'il y a un diagramme :*

$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle m'} \\ G & \xleftarrow{\ l'\ } & D & \xrightarrow{\ r'\ } & H \end{array}$$

tel que la partie gauche du diagramme est la déconnection de m vis-à-vis de k, et la partie droite un carré canonique.

Donc, suivant les théorèmes 10 and 11, un pas de réécriture correpond à un *double pushout* dans la catégorie des graphes.

Proposition 3 (Descriptions des noeuds) *En suivant la lexicographie de la définition 71, les représentants des classes d'équivalences des noeuds de $\mathcal{N}_R + \mathcal{N}_D$ peuvent être choisis de telle manière à ce que :*

$$\begin{array}{l} \mathcal{N}_H^{\Omega} = (\mathcal{N}_G^{\Omega} - m(\mathcal{N}_L^{\Omega})) + \mathcal{N}_R^{\Omega} \text{ et} \\ \mathcal{N}_H^{\mathcal{X}} = \mathcal{N}_G^{\mathcal{X}} + (\mathcal{N}_R^{\mathcal{X}} - r(\mathcal{N}_L^{\mathcal{X}})) \text{ et} \\ \mathcal{N}_H^{R} = r'(\mathcal{N}_D^{R}) \cup m'(\mathcal{N}_R^{R}) \end{array}$$

4.3 Caractérisation catégorique des ramasse-miettes

Un noeud dans un graphe enraciné est *atteignable* si c'est ce noeud fait partie des descencants de la racine ; Les noeuds non atteignables forment les *miettes* du graphe. Nous allons traiter le problème de la récupération des miettes (qui correspond dans la vie réelle à désalouer des cellules quand elles ne sont plus atteignables, pour les réutiliser) dans ses différents aspects : soit l'effacement des noeuds non atteignables ou bien leur réutilisation ; nous considérons aussi le marquage des noeuds non atteignables, qui constitue une étape importante du processus de tout ramasse-miettes.

4.3.1 La suppression des miettes est un adjoint à droite

On note par **Gr** la catégorie des graphes enracinés. La catégorie des graphes (sans racines) sera dénotée par $\mathbf{Gr}_0$, quand nous écrivons "graphe" nous sous-entendons implicitement "graphes enracinés", à moins que le contraire soit explicitement dit.

Définition 72 (Noeuds atteignables) *Les* noeuds atteignables *d'un graphe sont récursivement définis comme suit : une racine est accessible, et les successeurs d'un noeud accessible sont des noeuds accessibles*

Exemple 8 *Le graphe G défini dans l'exemple 4, a comme noeuds atteignables m, n, o, p, q, r. Les noeuds s, t sont considérés comme des miettes, tout commme les arrêtes sortant de ces noeuds.*

Définition 73 (Graphe atteignable) *Un* graphe atteignable *est un graphe dont tous les noeuds sont atteignables.*

Les graphes atteignables et les homomorphismes entre eux forment une sous-catégorie pleine de **Gr**, appelée la *catégorie des graphes atteignables* **RGr**. Soit V le foncteur d'inclusion :

$$V : \mathbf{RGr} \to \mathbf{Gr} .$$

Définition 74 (Sous-graphe maximal atteignable) *Le* sous-graphe maximal atteignable de G *est le graphe $\Lambda(G)$ tel que $\mathcal{N}_{\Lambda(G)}$ est l'ensemble des noeuds atteignables de G, $\mathcal{N}^R_{\Lambda(G)} = \mathcal{N}^R_G$, $\mathcal{N}^\Omega_{\Lambda(G)} = \mathcal{N}_{\Lambda(G)} \cap \mathcal{N}^\Omega_G$, et $\mathcal{L}_{\Lambda(G)}$, $\mathcal{S}_{\Lambda(G)}$ sont les restrictions de $\mathcal{L}_G$, $\mathcal{S}_G$ à $\mathcal{N}_{\Lambda(G)}$.*

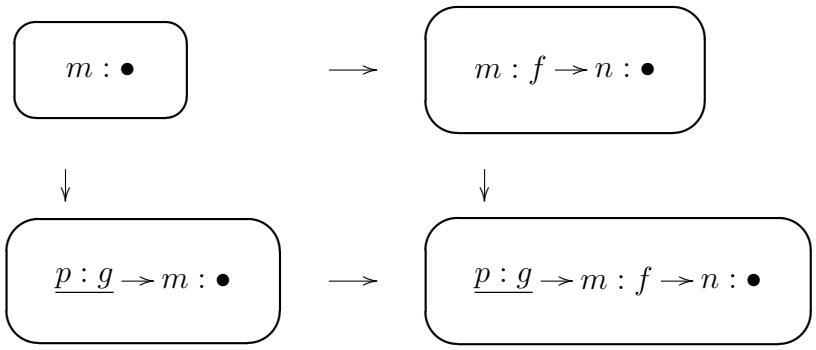
Comme un homomorphisme de graphe $\varphi : G \to H$ préserve les racines et les successeurs, il préserve aussi les noeuds atteignables. On peut donc le restreindre aux sous-graphes maximaux atteignables :

Définition 75 (Foncteur Ramasse-miettes) *Le* ramasse-miettes *est le foncteur :* Λ : **Gr** $\to$ **RGr** *qui associe chaque graphe* G *à son sous-graphe maximal atteignable* $\Lambda(G)$ *et chaque homomorphisme de graphes* $\varphi : G \to H$ *à sa restriction* $\Lambda(\varphi) : \Lambda(G) \to \Lambda(H)$.

Il est clair que le foncteur composé $\Lambda \circ V$ est l'identité sur **RGr** : un graphe atteignable n'est pas modifié par le ramasse-miettes.

Proposition 4 (Le ramasse-miettes est un adjoint à droite) *Le foncteur ramasse-miettes* Λ *est l'adjoint à droite du foncteur d'inclusion.* V.

Le foncteur de ramasse-miettes Λ : **Gr** $\to$ **RGr** n'est pas un adjoint à gauche. En effet, un adjoint à gauche préserve les co-limites, alors que le foncteur Λ ne préserve pas les pushouts, comme l'exemple suivant le montre :



Quand on applique Λ à ce carré, les graphes de la ligne supérieure seront des graphes vides, alors que les graphes de la ligne inférieure resteront identiques. Le diagramme résultant n'est plus un pushout de la catégorie **RGr**, car l'étiquette f et le noeud $n : \bullet$ surgissent de nulle part.

4.3.2 Le marquage des noeuds atteignables est un adjoint à gauche

Définition 76 (Graphes marqués) *Un* graphe *marqué est un graphe* M *avec un ensemble* $\mathcal{N}_M^\star \subseteq \mathcal{N}_M$ *de* noeuds marqués*, tel que chaque racine est marquée et chaque successeur d'une racine marquée est un noeud marqué (ainsi tous les noeuds atteignables d'un graphe marqué sont marqués mais des noeuds non atteignables peuvent également être marqués). Un homomorphisme de graphes marqués est un homomorphims de graphe préservant les noeuds marqués.*

Les graphes marqués et leurs homomorphismes forment la *catégorie des graphes marqués* $\mathbf{Gr}'$.

Soit $\Delta : \mathbf{Gr}' \to \mathbf{Gr}$ le foncteur qui oublie le marquage, et soit $\nabla : \mathbf{Gr} \to \mathbf{Gr}'$ le foncteur qui engendre un graphe marqué à partir d'un graphe, en marquant tous ses noeuds atteignables. On a alors le résultat suivant :

Proposition 5 (Marquage des noeuds atteignables) *Le foncteur de marquage des noeuds atteignables ∇ est l'adjoint à gauche du foncteur Δ, et cette adjonction est telle que $\Delta \circ \nabla \cong \mathrm{Id}_{\mathbf{Gr}}$.*

Définition 77 (Graphe marqué atteignable) *Un* graphe marqué atteignable *est un graphe marqué où tous les noeuds sont atteignables. En particuliers tous les noeuds d'un graphe marqué atteignable sont marquées.*

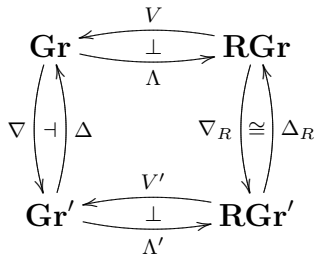
Les graphes marqués atteignables et les homomorphismes de graphes marqués forment une sous-catégorie pleine de $\mathbf{Gr}'$ appelée la *catégorie des graphes marqués atteignables*, $\mathbf{RGr}'$. On peut noter que $\mathbf{RGr}'$ est isomorphe à $\mathbf{RGr}$. Nous faisons la distinction entre les deux catégories car cela permet de donner une caractérisation catégorique claire de comment on effectue le ramassage des miettes. Soit V' le foncteur d'inclusion : $V' : RGr' \to \mathbf{Gr}'$. Comme dans la proposition 4, le foncteur d'inclusion V' a un adjoint à droite : $\Lambda' : \mathbf{Gr}' \to \mathbf{RGr}'$ qui est le foncteur ramasse-miettes pour les graphes marqués.

4.3.3 Détermination des miettes

En termes catégoriques, le fait que le marquage des noeuds accessibles peut être utilisé pour faire le ramassage des miettes est exprimé par le théorème 12 et la propostion 6 qui suit.

Symétriquement à l'adjonction $\nabla \dashv \Delta$, il y a une adjonction $\nabla_R \dashv \Delta_R$ où Δ_R : $\mathbf{RGr}' \to \mathbf{RGr}$ est le foncteur qui oublie le marquage et ∇_R : $\mathbf{RGr} \to \mathbf{RGr}'$ le foncteur qui engendre le graphe marqué à partir d'un graphe atteignable en marquant tous ses noeuds. En fait cette adjonction est un isomorphisme. $\mathbf{RGr}$ et $\mathbf{RGr}'$ ne sont pas identifiés car ils permettent de représenter une étape naturelle du ramassage de miettes : un ramasse-miettes commence par suspendre l'éxécution du processus en cours, ensuite il marque les cellules mémoires atteignables (∇). Ensuite, toutes les cellules non marquées sont désalouées (Λ') et finalement, le marquage est oublié (Δ_R).

Dans le diagramme suivant, les 4 paires d'adjonctions sont représentées.



Les foncteurs $\nabla \circ V$ et $V' \circ \nabla_R$ sont égaux, car ils associent tous deux un graphe atteignable H au graphe marqué composé de H et ayant toutes ses racines marquées :

$$\nabla \circ V = V' \circ \nabla_R : \mathbf{RGr} \to \mathbf{Gr'}$$

Le théorème 12 qui suit, donne une formalisation catégorique du processus de ramasse-miettes qu'on peut décomposer en trois étapes :

1. L'étape principale est d'engendrer librement le graphe marqué $\nabla(G)$ à partir du graphe d'origine G.
2. Ensuite, le graphe marqué accessible $\Lambda'(\nabla(G))$ est obtenu en supprimant tous les noeuds non marqués.
3. Enfin, le graphe atteignable $\Delta_R(\Lambda'(\nabla(G)))$ est obtenu en oubliant le marquage.

Selon le théorème 12, $\Delta_R(\Lambda'(\nabla(G)))$ est isomorphe à $\Lambda(G)$.

Théorème 12 (Elimination des miettes par marquage atteignable)

$$\Lambda \cong \Delta_R \circ \Lambda' \circ \nabla \ .$$

Donc, le foncteur d'élimination des miettes Λ peut être remplacé par $\Delta_R \circ \Lambda' \circ \nabla$. Cela signifie que l'étape la plus importante du rammassage des miettes est celle du marquage atteignable. En effet, l'application de Δ_R est "triviale", et l'application de Λ' sur l'image de ∇ consiste simplement à éliminer les noeuds non marqués.

Pour exprimer la récupération des miettes, soient respectivement $\mathcal{N}$ et $\mathcal{N}^\star$ les foncteurs de $\mathbf{Gr'}$ dans **Set** qui associent un graphe marqué à respectivement l'ensemble des noeuds de ce graphe, et l'ensemble des noeuds marqués de ce graphe. Il est possible de les combiner en un seul foncteur dont les valeurs sont dans la catégorie **SubSet** définie par : les objets de **SubSet** sont des paires d'ensembles (X, Y) telle que $Y \subseteq X$, et un morphisme est une

paire de fonction (f, g) de (X, Y) vers (X', Y') où $f : X \to X'$, $g : Y \to Y'$, et g est la restriction de f. $(\mathcal{N}, \mathcal{N}^\star) : \mathbf{Gr}' \to \mathbf{SubSet}$

Le complémentaire $X \setminus Y$ est l'ensemble des noeuds non-atteignables, on notera que l'opération de *complémentation*, qui associe (X, Y) à $X \setminus Y$, ne peut pas raisonnablement être étendue à un foncteur de **SubSet** dans **Set**.

Proposition 6 (Récupération des miettes) *La composition de ∇ avec $(\mathcal{N}, \mathcal{N}^\star)$, suivie de la complémentation, donne l'ensemble des noeuds non-atteignables.*

On peut donc exprimer la récupération des miettes par $(\mathcal{N}, \mathcal{N}^\star) \circ \nabla$ suivie par l'opération qui consiste à prendre complémentaire de cet ensemble.

Encore une fois, cela signifie que l'étape principale du ramassage des miettes est le marquage des noeuds atteignables.

4.3.4 Réécriture de graphes et ramassage des miettes

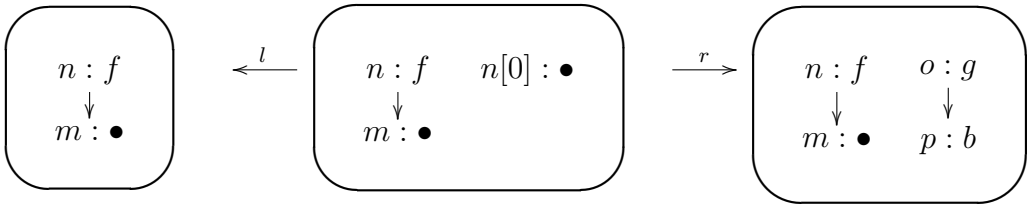
Une application directe de ce que nous venons de présenter est de pouvoir définir une notion de réécriture de graphes enracinées avec rammassage des miettes intégré. En effet, les adjoints à gauche préservent les pushouts. Il est donc possible d'appliquer le foncteur ∇ sur les doubles pushouts. Alors la composition $\Delta_R \circ \Lambda'$ nous donne le graphe réécrit sans noeuds non atteignables. Ce schéma peut s'appliquer à n'importe quel cadre utilisant le double pushout, nous nous concentrons sur la réécriture de graphe enracinés présentée en section 4.2.3.

Définition 78 (Pas de réécriture avec ramasse-miettes) *Soit p une règle de réécriture $(L \xleftarrow{l} K \xrightarrow{r} R)$ et $m : L \to G$ un filtrage, cohérents avec un kit de déconnection k de L. Alors G* se réécrit avec ramassage des miettes en P en utilisant la règle p *s'il existe un diagramme :*

$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle m'} \\ G & \xleftarrow{\ l'\ } & D & \xrightarrow{\ r'\ } & H \end{array}$$

dans lequel la partie est la déconnection de m vis-à-vis de k et la partie droite un carré canonique, et $P = V(\Delta_R(\Lambda'(\nabla(H)))) = V(\Lambda(H))$.

Exemple 9 *Considérons la simulation d'une règle de réécriture de termes simple comme $f(x) \to g(b)$. Dans notre cadre cela peut s'implanter par le span s suivant :*

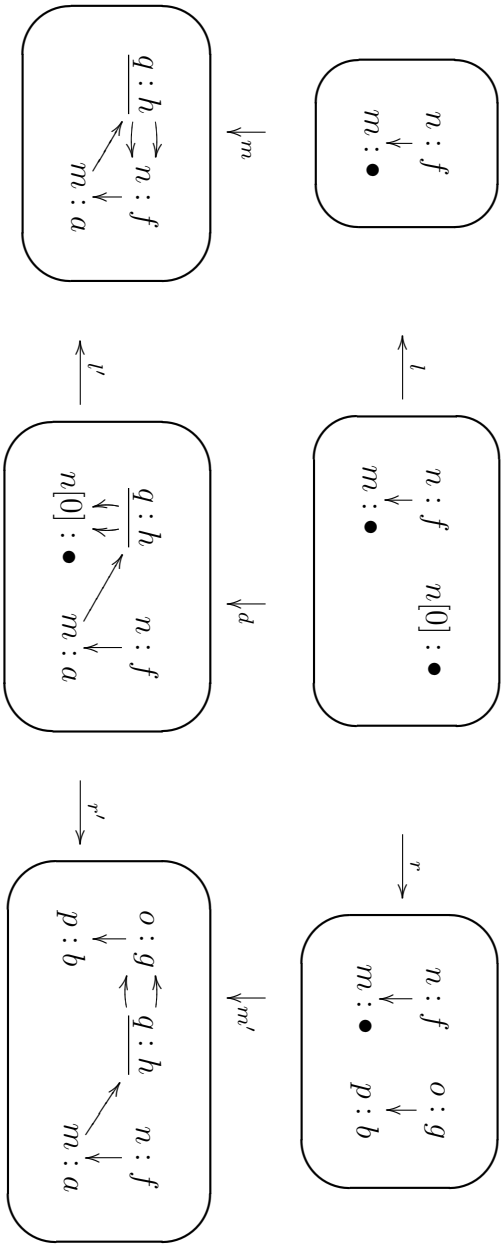


Où l, r sont les homomorphismes de graphes attendus avec $l(n[0]) = n$ et $r(n[0]) = o$. G_9 se réécrit en P_9 par l'utilisation de la règle s

$$G_9 = \underline{q : h} \rightleftarrows n : f \longrightarrow m : a$$

$$P_9 = p : b \longleftarrow o : g \rightleftarrows \underline{q : h}$$

En effet nous avons le double pushout suivant dans **Gr** *:*



Sur cet exemple $f(a)$ devient non atteignable à cause des redirections de pointeurs. Soit H_9 le graphe en bas à droite de ce double pushout. Alors, $\Lambda(H_9)$ est le graphe marqué, P_9 : $p * b \leftarrow o * g \overleftarrow{\leftarrow} \underline{q * h}$ où on écrit $n * f$ quand le noeud est marqué, et $n : f$ quand il ne l'est pas.

Le graphe suivant illustre bien le rôle joué par les racines dans le ramassage des miettes. Considérons G'_9 où la racine est maintenant n, ce graphe

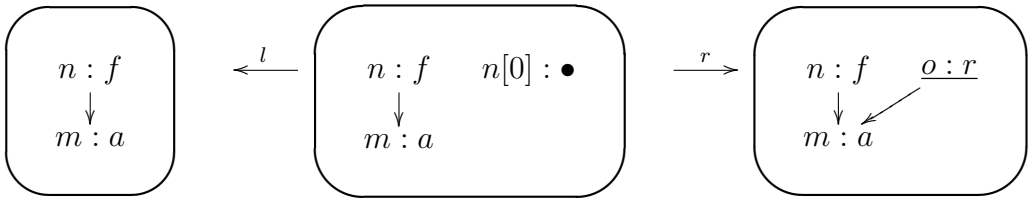
se réécrit en P'_9 avec :

$$G'_9 = \quad q : h \rightrightarrows \underline{n : f} \downarrow m : a$$

$$P'_9 = \quad \underline{o : g} \downarrow p : b$$

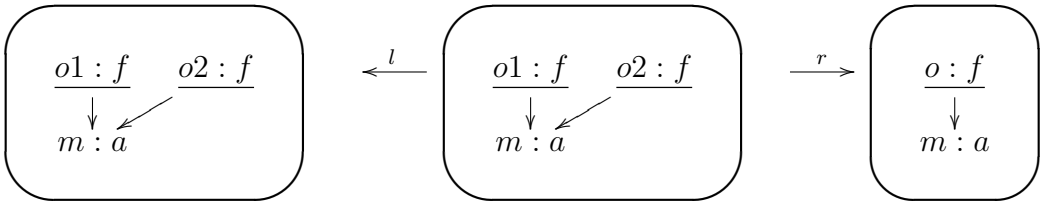
Ce simple exemple n'est pas possible à traiter dans des cadres utilisant le double pushout "standard" comme dans [DEP06] où les miettes ne peuvent pas être retirées.

Une autre propriété intéressante de la réécriture de graphes avec ramasse-miettes est le traitement des racines. De nouvelles racines peuvent être introduites par des règles comme :



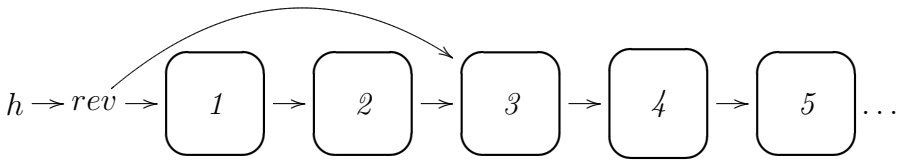
Dans laquelle $r(n[0]) = o$ et $l(n[0]) = m$. Cette règle ajoute une nouvelle racine $\underline{o : r}$.

Il est aussi possible de réduire le nombre de racines : pour cela il faut pouvoir associer deux racines portant les mêmes étiquettes. On peut par exemple considérer l'empan suivant :

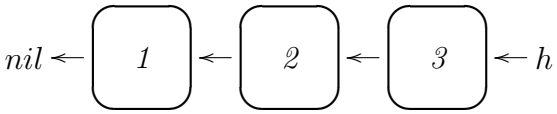


où $r(o1) = r(o2) = o$. On peut noter qu'à cause de l'hypothèse d'injectivité sur le filtrage des noeuds étiquetés, la partie gauche du empan doit filtrer deux racines différentes. De même on peut remarquer que l'hypothèse d'injectivité sur le morphisme r ne concerne que les noeuds non étiquetés, donc $o1, o2$ peuvent être confondus dans un seul noeud o.

Exemple 10 *Considérons maintenant un exemple plus élaboré : le renversement en place d'une liste entre deux cellules données. Par exemple, étant donné le graphe suivant :*



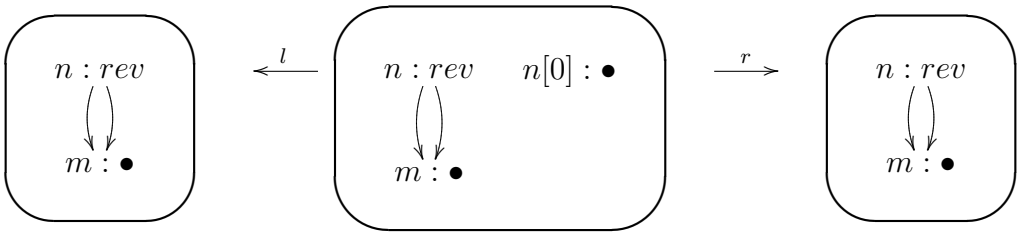
nous cherchons à obtenir le résultat suivant :



Potentiellement (s'il n'est pointé par rien d'autre), le reste du graphe doit être éliminé car il devient non accessible.

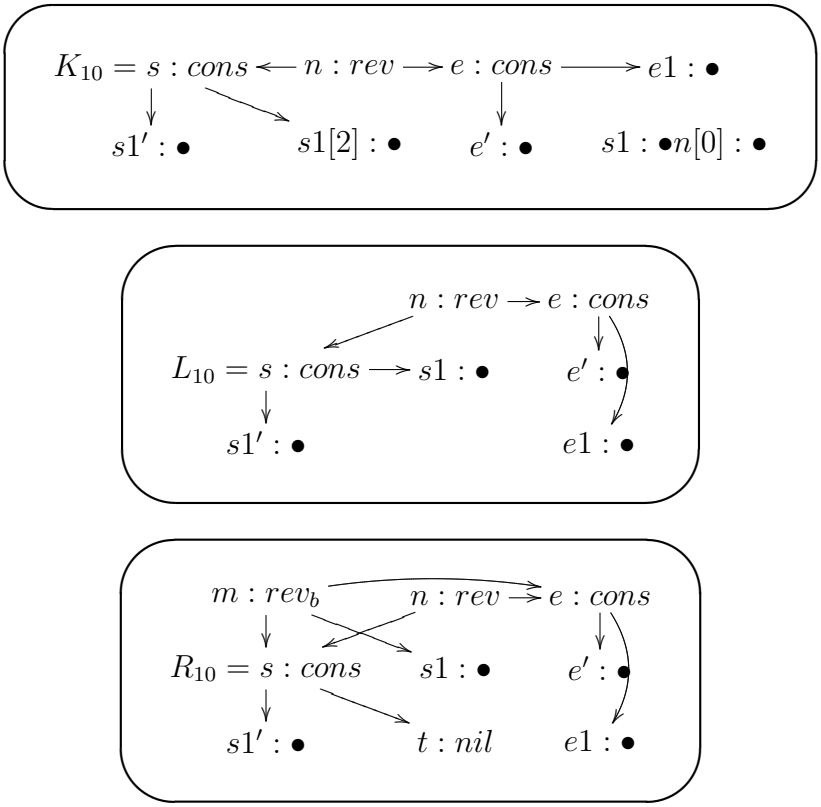
rev est défini par quatre règles. De plus, on peut observer que le programmeur n'a pas besoin de prendre un soin particulier pour traiter le cas des données qui ne sont plus accessibles : le traitement de celles-ci est entièrement automatique.

La première règle est pour les cas triviaux (quand les premières et dernières cellules sont égales) et s'implante de la façon suivante :



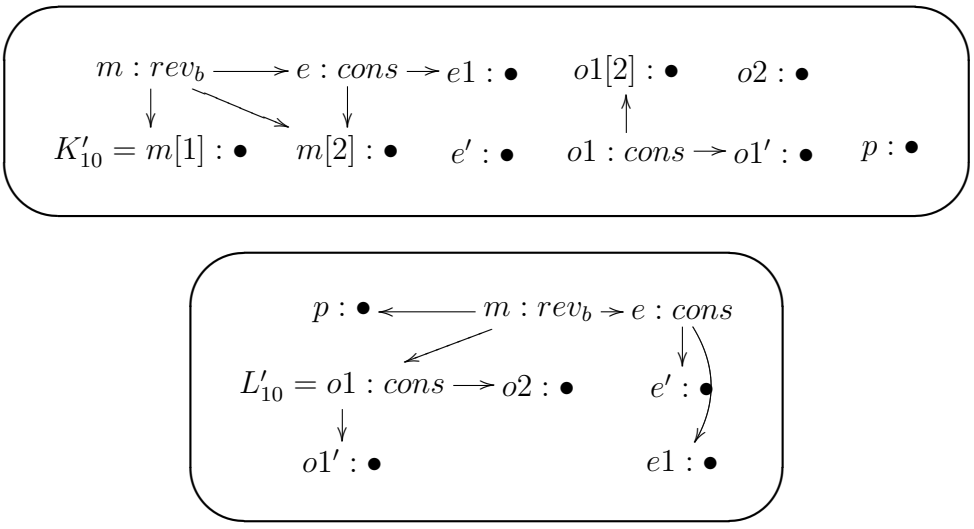
L'objectif de cette règle consiste simplement à faire une redirection de n vers m ($r(n[0]) = m$ et $l(n[0]) = n$), donc le noeud $n : rev$ va devenir inaccessible après le pas de réécriture (plus rien ne peut pointer sur ce noeud à cause de la redirection globale).

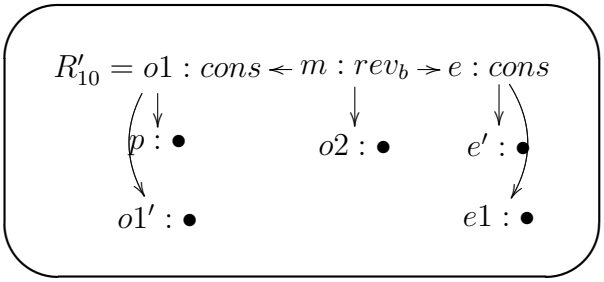
La deuxième règle est classique et introduit une fonction auxiliaire rev_b de trois paramètres qui effectue la réécriture sur la liste. Les deux premiers paramètres de rev_b enregistre la paire de cellules courante à inverser (la cellule actuelle et la précédente), le dernier paramètre enregistre la cellule sur lequel le calcul doit s'arrêter. Pour cela on utilise l'empan $L_{10} \leftarrow K_{10} \rightarrow R_{10}$ où :



où $n[0] : \bullet$ *est associé à* n *dans* L_{10} *et à* m *dans* R_{10}, $s1[2]$ *est associé à* t *dans* R_{10}.

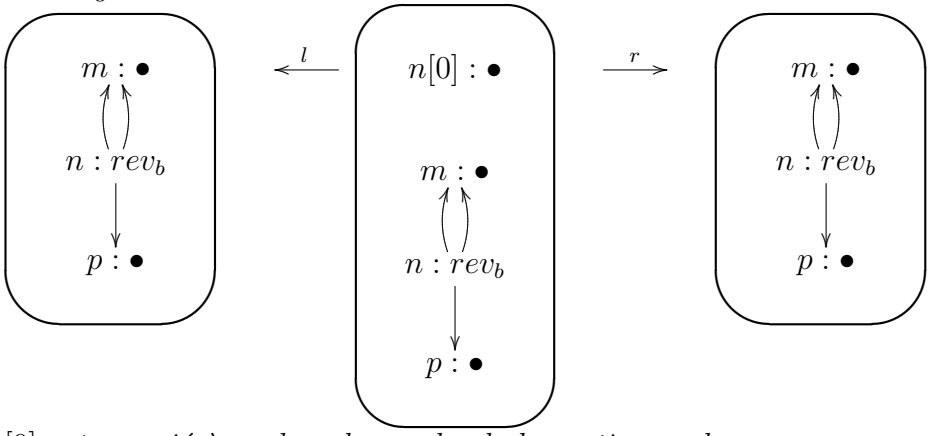
Le cas de récurrence de rev_b *est donné par la règle* $L'_{10} \leftarrow K'_{10} \rightarrow R'_{10}$, *où :*





dans laquelle $m[1], m[2]$ *sont respectivement associés à* $o1, o2$ *et* $o1[2]$ *est associé à* p *dans* R'_{10}*. Cette règle déconnecte le premier des deux paramètres de* rev_b *(*$m[1], m[2]$*) pour les faire progresser le long de la liste (*p *est remplacé par* $o1$*, et* $o1$ *par* $o2$*). Cette règle permet aussi de rediriger localement l'arrête de la liste qu'on doit inverser (la deuxième arrête de* $o1$*) à la cellule précédente (noeud* p*). On rappelle qu'à cause de l'hypotèse d'injectivité du filtrage sur les noeuds étiquetés, il est obligatoire que les noeuds* $o1$ *et* e *soient bien, en effet, des noeuds différents. Il n'y a donc pas de confusion possible avec le cas d'arrêt.*

Le cas d'arrêt pour rev_b *est similaire à celui de rev et revient juste à une redirection globale :*



où $n[0]$ *est associé à* m *dans le graphe de la partie gauche.*

4.3.5 Travaux reliés

Quand la réécriture est définie de manière algorithmique comme dans [BvEG+87], il n'y a pas vraiment de problème de fuite de mémoire (qui peut être gérée directement au niveau de la règle). Cependant ce n'est pas le cas pour les approches catégoriques. Dans la section 8 de [Ban94], Banach discute du problème des noeuds qui ne sont plus accessibles d'une manière abstraite. Il considère ce qu'il appelle "garbage retention", c'est à dire que les noeuds inaccessibles ne sont pas supprimés mais ne doivent plus être utilisés par le

processus de réécriture (ie ne doivent plus êtres filtrés). Dans [Bro91], Van den Broek aborde lui aussi le problème engendré par les noeuds inaccessibles mais dans le cadre des réécritures basées sur un simple pushout. Il introduit la notion de graphe propre qui peut se définir informellement par : un graphe propre est un graphe dans lequel une partie appelée "garbage" ne peut pas être atteinte par des noeuds qui ne sont pas du "garbage". La relation de réécriture est définie comme une relation binaire entre graphes propres. C'est à dire qu'une règle ne peut être utilisée que si le graphe résultant est propre, ce qui revient à la rétention pratiquée par Banach.

A notre connaissance il n'existe pas d'autre caractérisation catégorique de pas de réécriture prenant en compte l'élimination des parties non atteignables.

4.4 Gestion de la mémoire et réécriture de graphes

Dans la section précédente nous avons vu une approche permettant de supprimer les noeuds inaccessibles dans un formalisme de type double-pushouts. Nous allons maintenant voir comment donner la possibilité au programmeur de gérer directement la mémoire, que ce soit pour cloner ou éliminer (qui correspondra à cloner 0 fois) des parties du graphe. Pour cela on doit développer une notion de réécriture légèrement différente, on parle de sesqui-pushout [CHHK06] ou de pushout hétérogène [DEP09].

4.4.1 Graphes considérés

Dans cette section nous ne considérerons que des graphes sans racines. Nous introduisons quelques outils supplémentaires.

Définition 79 (Foncteur noeud) *Le* foncteur noeud $|-|$: **Gr** $\rightarrow$ **Set** *associe à chaque graphe* $G = (\mathcal{N}, \mathcal{D}, \mathcal{L}, \mathcal{S})$ *l'ensemble de ses noeuds* $|G| = \mathcal{N}$ *et chaque morphisme* $g : G \rightarrow H$ *à la fonction sous-jacente sur les noeuds* $|g| : |G| \rightarrow |H|$.

Comme le foncteur noeud est fidèle nous noterons souvent g au lieu de $|g|$. Cela signifie qu'un morphisme de graphe est uniquement déterminé par la fonction sous-jacente sur les noeuds. La fidélité du foncteur noeud implique qu'un diagramme sur les graphes est commutatif si et seulement si le diagramme commutatif sur les ensembles de noeuds est commutatif.

Nous introduisons une sorte de morphisme de graphe faible, les notions de *fonctions graphiques* et de *fonctions strictement graphiques*. Ces fonctions seront utilisées dans la section 4.4.2 pour mettre en relation les graphes lors d'un pas de réécriture.

Définition 80 (Fonction graphique) *Soient G et H deux graphes et $\gamma : |G| \to |H|$ une fonction sur leurs noeuds. Pour chaque noeud n de G, γ est* graphique en *n si soit n n'est pas étiqueté, soit n et $\gamma(n)$ sont étiquetés, $\mathcal{L}_H(\gamma(n)) = \mathcal{L}_G(n)$ et $\mathcal{S}_H(\gamma(n)) = \gamma^*(\mathcal{S}_G(n))$. De plus γ est* strictement graphique en *n si soit à la fois n et $\gamma(n)$ ne sont pas étiquetés soit les deux sont étiquetés. $\mathcal{L}_H(\gamma(n)) = \mathcal{L}_G(n)$ et $\mathcal{S}_H(\gamma(n)) = \gamma^*(\mathcal{S}_G(n))$. Pour chaque ensemble de noeuds Γ de G, γ est* graphique (resp. strictement graphique) en Γ *si γ est graphique (resp. strictement graphique) sur chaque noeud de Γ.*

Exemple 11 *Considérons les graphes G_1 et G_2 suivants :*

Soit $\gamma : |G_1| \to |G_2|$ la fonction définie par $\gamma = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d\}$. Il est aisé de vérifier que γ est strictement graphique en $\{1, 3\}$, est graphique, mais pas strictement, en $\{1, 2, 3\}$, et n'est pas graphique en $\{1, 2, 3, 4\}$.

Il est à noter que la propriété d'être graphique (resp. strictement graphique) et Γ implique les successeurs des noeuds de Γ, qui eux peuvent ne pas être dans Γ. De plus, il est clair qu'une fonction $\gamma : |G| \to |H|$ définie de manière sous-jascente un morphisme de graphe $g : G \to H$ si et seulement elle est graphique en $|G|$.

Lemme 8 *Soient G, H, H' des graphes et soient $\gamma : |G| \to |H|$, $\gamma' : |G| \to |H'|$, $\eta : |H| \to |H'|$ des fonctions telles que $\gamma' = \eta \circ \gamma$. Soit Γ un ensemble de noeuds de G. Si γ est strictement graphique en Γ et γ' est graphique en Γ, alors η est graphique en $\gamma(\Gamma)$.*

4.4.2 Réécriture avec clonage

Le nouveau type de règle que nous allons considérer peut être intuitivement décrit de la façon suivante : une règle aura une partie gauche, qui est un graphe L et une partie droite qui est un autre graphe R. Un pas de réécriture consiste à remplacer une instance de L dans un certain graphe G par H une instance de R.

Si la notion de "remplacement" est claire pour les noeuds, tout comme pour les arrêtes qui connectent tout ce qui en dehors de l'instance de L dans G, il n'en est pas de même pour les arrêtes entrantes et sortantes du motif filtré. Quand un noeud p étiqueté de G en dehors de l'image de L a son i^{eme} successeur p' dans L, alors p doit avoir son i^{eme} successeur n' quelque part dans H. Pour préciser cela nous introduisons une fonction τ des noeuds de L dans les noeuds de R, et nous posons que $\tau(p') = n'$.

Symétriquement, quand un noeud p étiqueté de G dans l'image de L a un i^{eme} successeur p', alors on peut exiger qu'un noeud n' de H a le même label que p et a son i^{eme} successeur qui est soit p', s'il est en dehors de L soit $\tau(p')$ sinon ; on appelle n un τ-clone de p. Comme chaque noeud de L peut avoir un nombre arbitraire de clones (et aussi pas de clone du tout), et qu'un noeud de R ne peut pas être le clone de plus d'un noeud de L, on spécifie cela par une fonction partielle σ des noeuds de R vers les noeuds de L et qui associe à chaque clone de p le noeud p.

Les fonctions partielles sont dénotées par le symbole "$\rightharpoonup$", le domaine d'une fonction partielle σ, sera noté $\mathrm{Dom}(\sigma)$, et la composition de fonctions partielles sera traitée comme d'habitude.

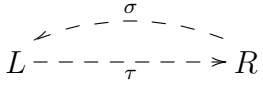
Le pas de réécriture est défini au moyen d'un simple *pushout hétérogène*, pour une règle T et un filtrage m, qui peut être construit à partir d'un pushout sur les ensembles.

Définition 81 (Clones) *Soient G et H des graphes et $\tau : |G| \to |H|$ une fonction. Alors $p \in |H|$ est un* τ-clone *de $q \in |G|$ si : p est étiqueté si est seulement si q est étiqueté, et $\mathcal{L}_H(p) = \mathcal{L}_G(q)$ et $\mathcal{S}_H(p) = \tau^*(\mathcal{S}_G(q))$.*

Définition 82 (Règle de réécriture) *Une* règle de réécriture *est un n-uplet (L, R, τ, σ) composé de deux graphes L et R, une fonction $\tau : |L| \to |R|$ et une fonction partielle $\sigma : |R| \rightharpoonup |L|$ telle que chaque noeud n dans le domaine de σ n'est pas étiqueté ou alors est un τ-clone de $\sigma(n)$.*

Un morphisme de règle de réécriture*, de $T = (L, R, \tau, \sigma)$ à $T_1 = (L_1, R_1, \tau_1, \sigma_1)$ est une paire morphismes de graphes (m, d) avec $m : L \to L_1$ et $d : R \to R_1$ tels que $|d| \circ \tau = \tau_1 \circ |m|$, $d(\mathrm{Dom}(\sigma)) \subseteq \mathrm{Dom}(\sigma_1)$ et $|m| \circ \sigma = \sigma_1 \circ |d|$ sur $\mathrm{Dom}(\sigma)$.*

Dans la suite, les illustrations sont faites soient dans la catégorie **Set** des ensembles ou alors dans un cadre hétérogène où les points sont des graphes, les flèches pleines sont des morphismes de graphes et les flèches en pointillé des fonctions sur les noeuds. Donc une règle de réécriture $T = (L, R, \tau, \sigma)$ sera représentée par :

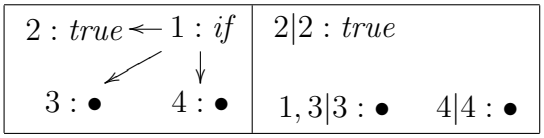


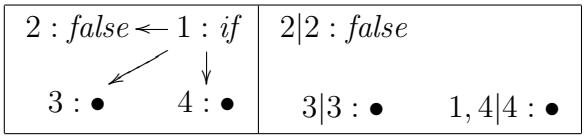
Afin de rendre la lecture des exemples plus aisée, une règle pourra aussi être représentée par

L	R

avec la convention que chaque noeud n dans R dans l'image de τ sera nommé $n = x, y, \ldots | w$ où $x, y, \ldots$ sont les noms des noeuds dans L tels que $\tau(x) = \tau(y) = \cdots = n$ et $\sigma(n) = w$. Quand n n'est pas dans l'image de τ alors on écrit $n = x|w$ avec x qui n'apparait pas dans L et quand $\sigma(n) = w$. Dans les deux cas "$|w$" est omis quand n n'est pas dans le domaine de σ.

Exemple 12 (if-then-else) *Les règles de réécriture suivantes permettent de définir l'opérateur "if-then-else" et font qu'il se comporte comme dans les langages impératifs.*

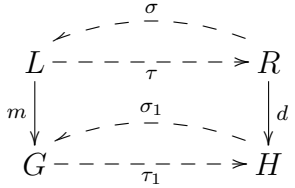




La définition de τ asure que l'expression "if-then-else" est remplacée par sa valeur $\tau(3) = 1, 3|3$ (resp. $\tau(4) = 1, 4|4$). La définition de σ indique que la valeur du "if-then-else" est son second (resp. troisième) argument en le spécifiant par $\sigma(1, 3|3) = 3$ (resp. $\sigma(1, 4|4) = 4$). Notons que si σ était définie comme la fonction vide, l'expression "if-then-else" s'évaluerait en un noeud non étiqueté. Notons aussi que, dans les deux parties droites des règles, le noeud 2 est dans le domaine de σ. Donc si le noeud 2 est partagé, les arrêtes incidentes ne seront pas modifiées après le pas de réécriture. Enfin, remarquons qu'il n'y pas de problèmes liés à l'injectivité (voir les conditions du filtrage dans la définition 85) et les trois branches du if-then-else peuvent être collapsées suivant n'importe quelle combinaison (potentiellement les noeuds 2,3,4 peuvent filtrer un seul noeud).

On peut noter que chaque morphisme de graphe $t : L \to R$ détermine si une règle de réécriture dans laquelle $\tau = |t|$ et σ n'est définie nulle part. Dans ce cas, pour chaque morphisme de graphe $m : L \to G$, le pushout de t et m dans la catégorie **Gr**, quand il existe, est l'objet initial dans la catégories des cônes sur t et m. Nous adaptons cette définition pour n'importe quelle règle de réécriture $T = (L, R, \tau, \sigma)$ et n'importe quel morphisme de graphe $m : L \to G$.

Définition 83 (Cône hétérogène) *Soient $T = (L, R, \tau, \sigma)$ une règle de réécriture et $m : L \to G$ un morphisme de graphe. Un* cône hétérogène sur T et m *est un n-uplet (H, τ_1, d, σ_1) composé d'un graphe H, d'une fonction $\tau_1 : |G| \to |H|$, d'un morphisme de graphes $d : R \to H$ et d'une fonction partielle $\sigma_1 : |H| \rightharpoonup |G|$ telle que $T_1 = (G, H, \tau_1, \sigma_1)$ est une règle de réécriture, $(m, d) : T \to T_1$ est un morphisme de règles de réécriture, τ_1 est graphique sur $|G| - |m(L)|$ et n_1 est un τ_1-clone de $\sigma_1(n_1)$ pour chaque n_1 dans le domaine de σ_1.*



Un morphisme de cônes hétérogènes sur T et m, *disons $h : (H, \tau_1, d, \sigma_1) \to (H', \tau'_1, d', \sigma'_1)$, est un morphisme de graphe $h : H \to H'$ tel que $|h| \circ \tau_1 = \tau'_1$, $h \circ d = d'$, $h(\mathrm{Dom}(\sigma_1)) \subseteq \mathrm{Dom}(\sigma'_1)$ et $\sigma'_1 \circ |h| = \sigma_1$ sur $\mathrm{Dom}(\sigma_1)$.*
Cela définit la catégorie $\mathbf{C}_{T,m}$ des cônes hétérogènes sur T et m.

Définition 84 (Pushout Hétérogène) *Soit $T = (L, R, \tau, \sigma)$ une règle de réécriture et $m : L \to G$ un morphisme de graphes. Un* pushout hétérogène de T et m *est un objet initial dans la catégorie $\mathbf{C}_{T,m}$ des cônes hétérogènes sur T et m.*

Quand un pushout hétérogène existe, son initialité implique qu'il est unique à isomorphisme de cônes hétérogènes près. L'existence d'un tel pushout, sous condition d'injectivité de m, est statuée dans le théorème 13.

Définition 85 (Filtrage) *Un* filtrage *vis-à-vis d'une règle de réécriture $T = (L, R, \tau, \sigma)$ est un morphisme de graphes $m : L \to G$ tel que si $m(p) = m(p')$ pour des noeuds distincts p et p' dans L alors $\tau(p)$ et $\tau(p')$ sont dans $\mathrm{Dom}(\sigma)$ et $m(\sigma(\tau(p))) = m(\sigma(\tau(p')))$ dans G.*

Proposition 7 *Soit $T = (L, R, \tau, \sigma)$ une règle de réécriture et $m : L \to G$ un filtrage vis-à-vis de T. Alors le pushout de τ et $|m|$ dans* **Set** *:*

$$\begin{array}{ccc} |L| & \xrightarrow[\tau]{} & |R| \\ {\scriptstyle |m|}\downarrow & & \downarrow{\scriptstyle \delta} \\ |G| & \xrightarrow[\tau_1]{} & \mathcal{H} \end{array}$$

satisfait $\mathcal{H} = \tau_1(\Gamma) + \delta(\Delta) + \delta(\Sigma)$ où $\Gamma = |G| - |m(L)|$, $\Sigma = \mathrm{Dom}(\sigma)$, $\Delta = |R| - \Sigma$ et : la restriction de $\tau_1 : \Gamma \to \tau_1(\Gamma)$ est bijective, la restriction

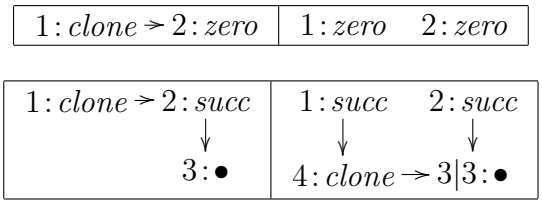
de $\delta : \Delta \to \delta(\Delta)$ est bijective, et la restriction de $\delta : \Sigma \to \delta(\Sigma)$ est telle que si $\delta(n) = \delta(n')$ pour des noeuds distincts n et n' dans Σ alors $m(\sigma(n)) = m(\sigma(n'))$ dans G. De plus, il existe une unique fonction partielle $\sigma_1 : \mathcal{H} \rightharpoonup |G|$ dont le domaine est $\delta(\Sigma)$ telle que $|m| \circ \sigma = \sigma_1 \circ \delta$.

Proposition 8 *Soit $m : L \to G$ un filtrage vis-à-vis de $T = (L, R, \tau, \sigma)$. Le pushout de τ et $|m|$ dans* **Set***, avec σ_1 tel que dans la proposition 7, définit un cône hétérogène sur T et m.*

Théorème 13 *Etant donnés une règle $T = (L, R, \tau, \sigma)$ et un filtrage $m : L \to G$ vis-à-vis de T, le cône hétérogène $(m, d) : T \to T_1$ sur T et m défini dans la proposition 8 est un puhsout hétérogène de T et m.*

Définition 86 (Pas de réécriture) *Etant donnés une règle de réécriture $T = (L, R, \tau, \sigma)$ et un filtrage $m : L \to G$ vis-à-vis de T, le* pas de réécriture *correspondant construit le morphisme de graphe $d : R \to H$, obtenu à partir du pushout hétérogène de T et m.*

Exemple 13 (Clonage de structures de données) *Nous donnons deux règles pour cloner des entiers, encodés avec succ et zero. Il est en fait possible de généraliser ces règles à n'importe quelle structure de données.*



La première règle gère le clonage de zero. Etant donné un filtrage $m : L \to G$, comme $\tau(1) = 1$ cette règle transforme toutes les arrêtes dans G ayant comme but $m(1 : clone)$ en des arrêtes de H ayant pour but $d(1 : zero)$, et comme $\tau(2) = 2$ les arrêtes de G avec but $m(2 : zero)$ restent "inchangées" dans H, dans le sens où leur but est $d(2 : zero)$.

La deuxième règle permet de gérer le clonage des entiers non-nuls. Comme $\sigma(3|3) = 3$, l'étiquette et les successeurs du noeud $m(3)$ dans G seront les mêmes que l'étiquette et les successeurs du noeud $d(3|3)$ dans H. On notera que dans ce cas, il ne serait pas permi de définir $\sigma(4) = 2$ car le noeud 4 de R est étiqueté par clone et le noeud 2 de L est étiqueté par succ, ce qui contredirait la condition pour les τ-clones.

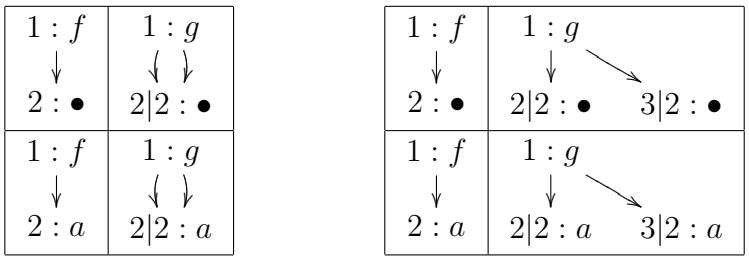
Nous représentons un pas de réécriture de G à H en utilisant la règle (L, R, τ, σ) comme ci dessous, en utilisant les mêmes conventions de notation

que précédemment concernant le nom des noeuds dans la partie droite de la règle. Quand le filtrage, m, est injectif on écrit $m(p) = p$.

L	R
G	H

Exemple 14 *Considérons la règle de réécriture de terme* $f(x) \to g(x, x)$. *Dans notre cadre, cette règle peut être traduite de différentes manières selon la représentation de* $g(x, x)$ *sous forme de terme-graphe et selon la manière dont les fonctions* τ *et* σ *sont définies.*

Par exemple voilà deux règles différentes et leurs applications au terme-graphe $1 : f(2 : a)$, *avec le filtrage préservant le nom des noeuds. La deuxième règle produit deux clones de* $2 : a$ *in* G *:* $2|2 : a$ *et* $3|2 : a$ *dans* H



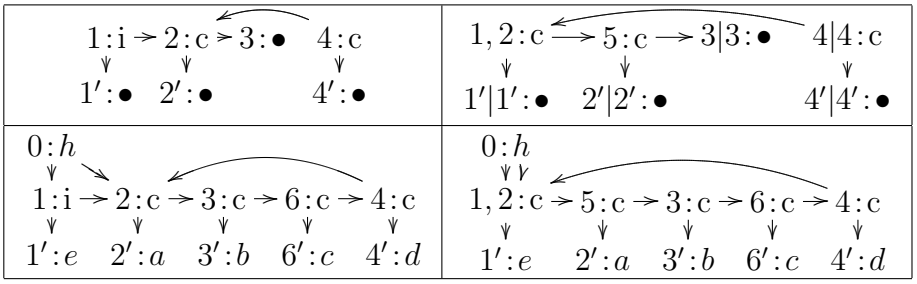
Comme la réécriture de terme-graphes avec pushout hétérogène repose sur un pushout des noeuds sous-jascents (proposition 7), et également car la condition pour le filtrage (definition 85), il y a une relation évidente entre la taille des terme-graphes réécris et la taille du terme-graphe original pour chaque pas de réécriture. De plus cette relation peut être inférée au niveau des règles. Il est donc possible d'analyser l'utilisation mémoire d'un programme simplement en inspectant ses règles. La proposition 9, où $\sharp$ dénote le cardinal d'un ensemble établit formellement ce résultat. Donc si on choisit comme mesure de la mémoire utilisée la taille d'un terme-graphe (donc cette fois-ci en laissant de côté les problèmes liés à la non-atteignabilité), alors il est possible de calculer statiquement, pour chaque règle, la taille de mémoire requise ou rendue par un pas de réécriture.

Proposition 9 *Soit* $T = (L, R, \tau, \sigma)$ *une règle de réécriture,* $m : L \to G$ *un filtrage vis à vis de* T *et* $d : R \to H$ *le résultat du pas de réécriture. Alors,* $\sharp|H| - \sharp|G| = \sharp|R| - \sharp|L|$.

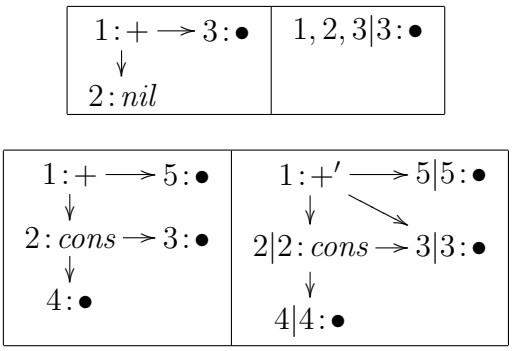
Exemple 15 (Insertion dans une liste circulaire) *Nous présentons une règle pour l'insertion d'un élément en tête d'une liste circulaire et dont la taille est plus grande que un. Dans la partie gauche de cette règle, le noeud*

2 *est la tête de la liste, et le noeud* 4 *est le dernier élément de la liste. Le pointeur du noeud* 4 *à la tête de la liste est redirigé de* 2 *(dans L) vers un nouveau noeud* $1, 2$ *(dans R). La définition de* τ *est telle que tous les pointeurs vers la tête de la liste seront redirigés de* 2 *vers* $1, 2$*. Nous montrons l'application de cette règle à une liste circulaire de 4 éléments.*

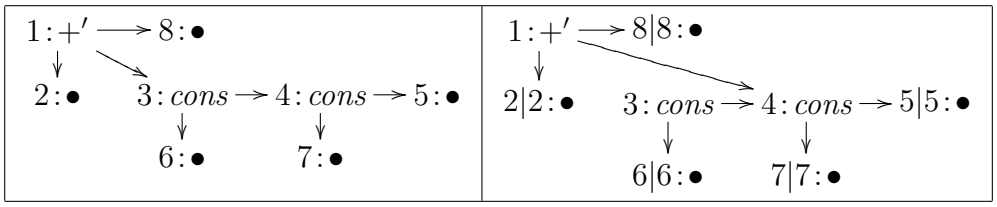
Nous notons c *et* i *pour, respectivement, cons et insert.*



Exemple 16 (Concaténation de listes chaînées) *Nous considérons maintenant l'opération "+" qui concatène, en place, deux listes chaînées. Les listes sont construites avec les constructeurs cons et nil.*

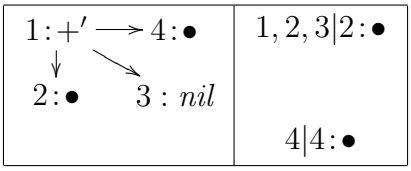


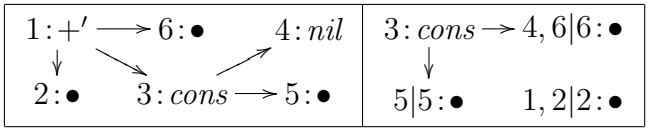
La première règle gère le cas de base, quand le premier argument est nil. La deuxième règle implante le fait que quand le premier argument est une liste non vide, alors il faut appeler un fonction intermédiaire "+′" d'arité 3. Le rôle de cette fonction est de parcourir la liste jusqu'à sa fin et de concaténer les deux listes en redirigeant le dernier pointeur de la première liste. L'argument additionnel de $+'$ *est un pointeur utilisé pour le parcours de liste et est initialisé sur la seconde cellule (quand la longueur de la liste est au moins 2) il parcourt ensuite la liste jusqu'à sa dernière cellule. Ce qui est implanté de la façon suivante :*



Les cas d'arrêts pour l'opération $+'$ sont implantés comme suit. Un cas est réservé quand la première liste est vide. Dans la deuxième règle, l'arrête $3 \to 4$ dans L est redirigée en $3 \to 4, 6|6$ dans R, qui se trouve être la tête de la deuxième liste à concaténer.

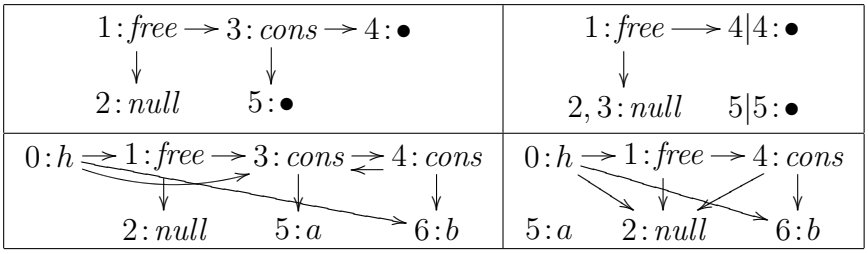
Le résultat global de l'opération $+'$ est le noeud $\tau(1) = 1, 2|2$, qui est la tête de la seconde liste.



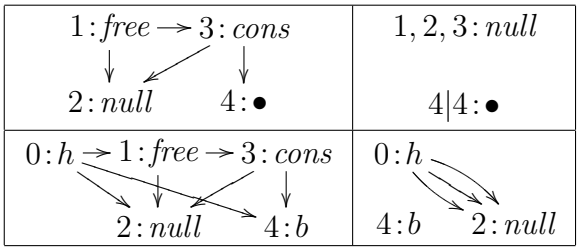


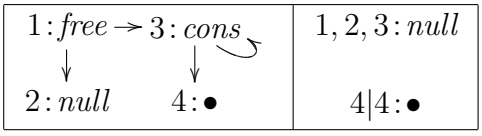
Exemple 17 (Libération de mémoire) *Dans cet exemple nous montrons comment nous pouvons libérer la mémoire utilisée par une liste circulaire (en fait nous libérons juste les cellules mémoires utilisées pour les noeuds "cons"). Comme nous considérons des graphes de termes dans lesquels chaque symbole de fonction a une arité fixée, il n'est pas possible de créer des pointeurs "pendants". Cette contrainte est exprimée par le fait que chaque noeud de la partie gauche L doit avoir une image en partie droite par τ. L'opération free a deux arguments. Le premier est un noeud particulier étiqueté par null, il est destiné à devenir le but des arrêtes qui pointaient sur les noeuds libérés (ce qui évite la création de pointeurs pendants dans le vide après libération). Le second argument de free est la liste des cellules restantes qu'il faut libérer.*

La règle définissant le comportement de l'opération free dans le cas d'une liste avec au moins deux éléments est donnée ci-après. Nous illustrons son application sur une liste de taille 2. Il est à noter que les pointeurs entrants dans les noeuds 3 et 5 sont redirigés vers 2 dans H.



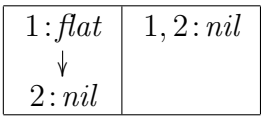
Pour les listes ne contenant qu'un élément, à cause de la condition d'injectivité sur le filtrage, il faut définir deux règles. La première règle spécifie le cas où le dernier élément de la liste est obtenu après libération d'autres éléments dans la liste. Nous illustrons ce pas de réécriture sur le graphe obtenu précédemment (à un renommage près des noeuds). La deuxième règle traite le cas spécial des listes de taille exactement un.

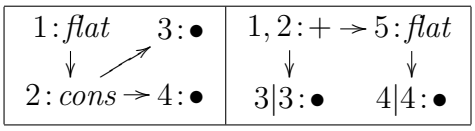




Exemple 18 (Analyse de l'utilisation de la mémoire) *Nous allons maintenant voir comment la proposition 9 peut être utilisée sur un programme qui applatit une liste de listes. Cet exemple ne peut pas être traité en utilisant un système de types pour l'analyse de l'utilisation de la mémoire comme l'ont proposé [HJ03]. En effet, un système de types avec types dépendants (incluant par exemple des types du genre "liste de taille n") serait nécessaire pour cela.*

Le programme consiste est formé des règles suivantes, la première est utilisée pour le cas de base (la liste vide) et la deuxième règle permet de gérer les listes de listes non-vides. Ici + est l'opération de concaténation et $+'$ sa fonction auxiliaire comme dans l'exemple 16.

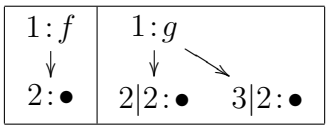




Donc l'utilisation de la mémoire pour l'applatissment d'une liste peut être analysée en considérant six règles : deux pour flat, deux pour + et deux pour $+'$. L'inspection des règles de récurrence pour les trois fonctions montre que le nombre de noeud reste inchangé. Le cas d'arrêt pour flat libère une case mémoire. Le cas d'arrêt pour + and $+'$, libère deux cellules mémoire. Maintenant en raisonnant simplement sur les règles invoquées pour l'évaluation de flat on voit que flat(ℓ) libère exactement $2|\ell| + 1$ cellules mémoire, où $|\ell|$ est la taille de la liste ℓ. En effet il y a un cas d'arrêt pour flat, et pour chaque élément de ℓ il y a un cas d'arrêt pour + ou un autre pour $+'$.

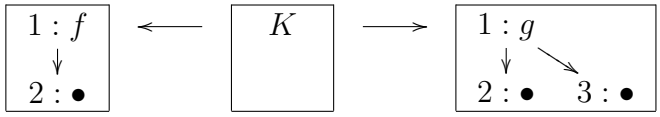
4.4.3 Travaux reliés

Le clonage est aussi une des caractéristiques des approches basées sur le sesqui-pushout [CHHK06]. Dans cette approche, une règle est un empan $L \leftarrow K \rightarrow R$ et l'application d'une règle à un graphe G peut s'illustrer de la même manière qu'un double pushout. Simplement le carré de gauche est un pullback, de plus c'est un pullback complément. Les graphes considérés dans [CHHK06] sont définis par $G = (V, E, \texttt{src} : E \rightarrow V, \text{tgt} : E \rightarrow V)$ où V et E sont respectivements les sommets et les arrêtes, les arrêtes sont définies par les fonctions `src` (source) et tgt (but). Il n'y a pas d'arité associée aux noeuds. Cette particularité tout comme le fait que le carré de gauche soit un pullback complément final marquent la différence entre cette approche et la notre. Selon l'approche du sesqui-pushout le clonage d'un noeud consiste à copier le noeud avec toutes ses arrêtes adjacentes (entrantes et sortantes). Dans notre approche un clone n'est copié qu'avec ses arrêtes sortantes. Considérons le terme-graphe $h(f(2:a), 2)$ dans lequel le sous-graphe $f(2:a)$ est supposé être transformé en $g(2, 3)$ avec les noeuds 2 et 3 qui sont des clones de $2:a$. Alors d'après notre réécriture, cette transformation peut être faite au moyen de la règle suivante



L'application de cette règle à $h(f(2:a), 2)$ produit le graphe $h(g(2:a, 3:a), 2)$. En utilisant une approche à la sesqui-pushout, nous aurions une règle de la

forme :

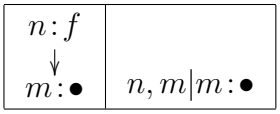


En effet, K doit encoder le clonage de l'instance du noeud 2 ainsi que le remplacement de f par g, donc K doit contenir au moins trois noeuds non étiquetés.

L'application de cette règle à $h(f(2:a),2)$ produit le graphe $h(g(2:a,3:a),2,3)$ dans leque h se retrouve avec trois arguments, car chaque clone $2:a$ et $3:a$ requiert le clonage de toutes les arrêtes entrantes.

Le clonage a aussi été étudié dans [DHJ+06]. Les auteurs considèrent des règles de réécriture de la forme $S := R$ où S est une étoile, c'est-à-dire que S est un noeud entouré par ses noeuds adjacents ainsi que les arrêtes qui les connectent. Les règles de réécritures qui font du clonage de noeud sont données dans la définition [DHJ+06, Def. 6]. Ces règles montrent comment une étoile peut être supprimée, conservée à l'identique ou bien copiée (clonée) plus d'une fois. Ici encore, à la différence de notre approche, le clonage ne prend pas en compte l'arité des noeuds et, comme dans le cas de l'approche à la sesqui-pushout, un noeud est copié avec toutes ces arrêtes entrantes et sortantes. Si on considère de nouveau le terme-graphe $h(f(2:a),2)$ et que l'on clone deux fois le noeud $2:a$, alors d'après [DHJ+06] nous obtenons le graphe $h(f(2:a,3:a),2,3)$ dans lequel h et f ont vu leur arité augmenter à cause des copies des arrêtes incidentes .

Un cadre catégorique dédié aux transformations de graphes cycliques à été proposé dans [CG97] où les auteurs, suivant [Pow89], proposent une représentation 2-catégorique de la réécriture de graphe. Le résultat permet presque une simulation opérationnelle complète de la réécriture de graphe définie dans [BvEG+87], mais il subsiste une différence lorsque l'on considère les redex ciruclaires. Par exemple, l'application de la règle $f(x) \to x$ sur le terme-graphe $n:f(n)$ produit le même graphe (i.e., $n:f(n)$) selon [BvEG+87] mais produit un noeud non-étiqueté, disons $p:\bullet$, suivant [CG97]. Selon notre définition de règles de réécriture, il est possible d'encoder exactement l'approche de [BvEG+87]. Il suffit simplement de dire que le noeud $n,m|m$ est un clone du noeud m dans la règle :



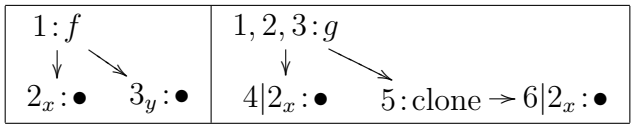
De manière générale, la réécriture de termes et de terme-graphes ne coïncident pas [Plu99]. Cependant, grâce au clonage, notre cadre permet de simuler la réécriture de terme de le cas des systèmes linéaire-gauche. En effet,

une règle de réécriture de terme $l \to r$ dans laquelle l est un terme linéaire (i.e., les variables de l n'ont qu'une seule occurrence dans l), peuvent être encodées en une règle (L, R, τ, σ) où $L = \texttt{l2L}(l)$ et $R = \texttt{r2R}(r)$ sont les terme-graphes correspondants respectivement à l et r, selon les transformations `l2L` et `r2R` définies ci-dessous.

Notons que la transformation de la partie droite prend en compte le clonage des instances de variables, cela arrive quand une partie droite n'est pas linéaire. Le but de τ, dans cette simulation de réécriture de terme, est d'indiquer le remplacement de la racine de L par celle de R, c'est à dire, $\tau(\texttt{root}(L)) = \texttt{root}(R)$. Les images via τ des noeuds restants ne sont pas signifiantes, donc $\tau(|L|) = \texttt{root}(R)$ est un choix possible pour la définition de τ. La fonction σ indique les parties que l'on doit cloner, cette fonction joue un rôle important en permettant d'encoder l'utilisation des variables dans la partie droite. Chaque occurrence d'une variable x dans r correspond a un noeud non-étiqueté $p : \bullet$ dans R. Dans ce cas, par définition des règles de réécritures, x apparaît dans l et donc un noeud non étiqueté correspondant $p_x : \bullet$ apparaît dans L, d'où nous pouvons affirmer que p est un clone de p_x par $\sigma(p) = p_x$. Maintenant nous donnons la définition des fonctions `l2L` et `r2R`.

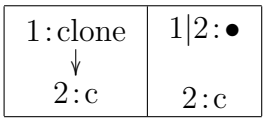
- Si c est une constante alors $\texttt{l2L}(c) = p : c$ et $\texttt{r2R}(c) = p : c$ avec p un noeud frais
- Si f est un symbole de fonction et les t_i sont des termes alors $\texttt{l2L}(f(t_1, \ldots, t_n)) = p : f(\texttt{l2L}(t_1), \ldots, \texttt{l2L}(t_n))$ et $\texttt{r2R}(f(t_1, \ldots, t_n) = p : f(\texttt{r2R}(t_1), \ldots, \texttt{r2R}(t_n))$ avec p un noeud frais.
- Si x est une variable alors
  - $\texttt{l2L}(x) = p_x : \bullet$ avec p_x un noeud frais.
  - $\texttt{r2R}(x) = p : \bullet$ pour la première occurrence de x dans r avec p un noeud frais et $\sigma(p) = p_x$, sinon $\texttt{r2R}(x) = q : \text{clone}(p : \bullet)$, avec les noeuds p et q qui sont frais et $\sigma(p) = p_x$.

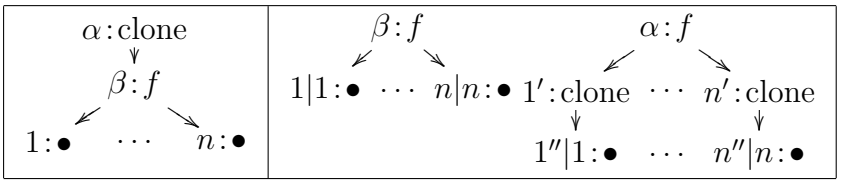
Par exemple, la règle de réécriture de termes $f(x, y) \to g(x, x)$ est traduite par :



L'application d'un filtrage m sur la partie droite r d'une règle de réécriture de terme, pour chaque variable x de la partie gauche construit autant de copies de l'instance $m(x)$ que le nombre d'occurrences de x dans r. L'explication de $q : \text{clone}(p : \bullet)$ dans la définition de $\texttt{r2R}(x)$ est que cela permet de simuler l'application du filtrage sur la partie droite $m(r)$. La fonction clone construit une copie de son argument, et peut être définie pour chaque symbole d'une

signature comme suit :





avec c une constante et f un symbole de fonction d'arité n. On écrit $\rightarrow^*_{\texttt{CLONE}}$ la relation de réécriture induite par les règles précédentes.

Proposition 10 *Soit $\mathcal{R}$ un système de réécriture de termes linéaire gauche construite sur la signature Ω. Soit $T(\mathcal{R})$ le système de réécriture de terme-graphe consitué de règles de réécriture qui définissent la fonction clone sur les symboles de fonction de Ω, ainsi que les transformations des règles $l \rightarrow r$ de R. Soit t un terme clos Si $t \rightarrow_{\mathcal{R}} t'$ alors il existe un terme-graphe G_1 tel que $\texttt{12L}(t) \rightarrow_{T(\mathcal{R})} G_1 \rightarrow^*_{\texttt{CLONE}} \texttt{12L}(t')$.*

Chapitre 5

Conclusion et perspectives

Je compte prolonger mes recherches selon deux axes principaux. Le premier est une continuation et une prolongation autour du thème de la non-interférence. Le second axe consiste en l'approfondissement des études que j'ai déjà mené dans le cadre de la théorie des langages de programmation quantique, et l'étude des abstractions utiles à la programmation dans un tel cadre.

Une approche unifiée pour la non-interférence

Le problème de la non-interférence est appelé à de nombreux développements. Il est en effet une des bases de la notion de confidentialité dans les systèmes informatiques. Cette problématique devient de plus en plus importante du fait d'une part de la multiplication des opérations numériques (compte bancaire, santé etc.) et d'autre part de l'apparition de ce qui est nommé le "cloud computing". De plus en plus d'opérations sont menées on ne sait où, par on ne sait quelles machines. Le problème de la confiance et de la gestion des informations sensibles est un enjeu sociétal important.

Une première piste de recherche consiste à explorer plus précisément la notion "d'interférences acceptables" et comment les gérer, comme nous l'avons présenté avec la notion de politique de confidentialité dan la section 2.2.2, dans un tel contexte. Des avancées récentes [Gen09] sur des processus de cryptages complètements homomorphes (où donc on peut calculer sur les données sans avoir à les décrypter) ouvrent de nouvelles perspectives de recherches sur la bonne notion d'interférences acceptables (car calculer sur des données cryptées peut conduire à des interférences qui ne sont en fait pas dangereuses).

Le problème de la confidentialité est encore mal exploré dans le domaine

de la réécriture de graphes : on peut le voir comme une approche théorique de la confidentialité en présence de pointeurs. La caractérisation du ramasse-miettes en termes catégoriques 4.3 (qu'on peut voir comme étant la suppression de ce qui n'interfère pas avec le reste du graphe) est un premier résulat dans ce sens.

Enfin, la logique de l'intrication que définie en section 3.3 est issue des recherches de type non-interférence dans un cadre avec pointeurs. Ce dernier point montre comment des techniques venant de champs différents (du classique vers le quantique dans ce cas, mais on peut penser aux algèbres de processus par rapport aux programmes impératifs / fonctionnels etc.) peuvent se révéler fructueuses. Un projet ambitieux serait donc de travailler à une approche unifiant ces différentes techniques (typage, interprétation abstraite, logique) dans différents modèles de calcul (réécriture de graphes, calcul quantique, algèbre de processus etc.) ou tout du permettant la lecture de certains résultats sous d'autres angles ce qui, comme je l'ai montré, amène des résultats originaux et inattendus.

Logique, typage et calcul quantique

Comme on a pu le constater dans le chapitre 3 de ce mémoire la physique quantique n'est pas du tout intuitive et c'est encore pire quand on se place du point de vue de la programmation quantique car il faut pouvoir utiliser ces particularités pour en obtenir un gain par rapport à la programmation classique. Les bonnes abstractions ne sont pas encore connues : il s'agit d'un vaste champs de recherche qui s'ouvre.

L'étude de système de typages pour les calculs quantiques est en plein essor. Pour l'instant il n'y a pas encore d'approche canonique pour ce genre de calcul. Un de mes objectifs est de donner une version quantique (ou linéaire algebrique [AD08]) du λ-cube classique [Bar91]. Une telle approche pourrait servir de squelette à des analyses basées sur le typage, comme ce fut le cas en classique [Pro00]. Elle pourrait aussi amener, en suivant les lignes de la correspondance de Curry-Howard, à la découvertes de nouvelles logiques dans le cadre quantique.

Bibliographie

[AB02] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proceedings of the* 29^{th} *Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 33 – 44, Portland, January 2002. ACM Press.

[Aba97] M. Abadi. Secrecy by typing in security protocols. In *Proc. of 3rd Theoretical Aspects of Computer Software, LNCS 1281*, pages 611–638. Springer, 1997.

[AD08] P. Arrighi and G. Dowek. Linear-algebraic lambda-calculus : higher-order, encodings, and confluence. In *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2008.

[AG05] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science*, 2005.

[AK96] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3-4), 1996.

[ALSU06] A. Aho, M. S. Lam, R. Stehi, and J. D. Ullman. *Compilers : Principles, Tools & Techniques*. Pearson Education lnc, 2nd edition edition, 1986, 2006.

[Ban94] R. Banach. Term graph rewriting and garbage collection using opfibrations. *Theoretical Computer Science*, 131 :29–94, 1994.

[Bar91] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2) :125–154, 1991.

[Bar96] A. Barber. Dual intuitionistic logic. Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Science, University of Edimburgh, 1996.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96 :217–248, 1992.

[BB97] S. Berardi and L. Boerio. Minimum information code in a pure functional language with data types. In P. de Groote, editor, *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*, volume 1210 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 1997.

[BC01] G. Boudol and I. Castellani. Non-interference for concurrent programs. In F. Orejas, P.G. Spirakis, and J. van Leeuwen, editors, *Proceedings of the 28^{th} International Colloquium on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *lncs*, pages 382–395., Cesena, July 2001. Springer Verlag.

[BC02] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1) :109 – 130, 2002. Special issue : "Merci, Maurice, A mosaic in honour of Maurice Nivat" (P.-L. Curien, Ed.).

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Coq'Art : The Calculus of Inductive Construction Series. Springer, 2004.

[BG96] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43 :1–50, 1996.

[BHY05] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, pages 280–293, 2005.

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[Bou97] G. Boudol. The pi-calculus in direct style. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 228–241, 1997.

[Bro91] P.M. Van Der Broek. Algebraic graph rewriting using a single pushout. In *TAPSOFT'91 : Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 493 of *LNCS*, pages 90–102. Springer, 1991.

[BvEG+87] H. Barendregt, M. van Eekelen, J. Glauert, R. Kenneway, M. J. Plasmeijer, and M. Sleep. Term graph rewriting. In *PARLE'87*, pages 141–158. LNCS 259, 1987.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, New York, 1977. ACM Press.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 493–501. ACM Press, 1993.

[CG97] A. Corradini and F. Gadducci. A 2-categorical presentation of term graph rewriting. In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science, 7th International Conference, CTCS '97, Santa Margherita Ligure, Italy, September 4-6, 1997, Proceedings*, volume 1290 of *Lecture Notes in Computer Science*, pages 87–105. Springer, 1997.

[CHHK06] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesquipushout rewriting. In *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 30–45, 2006.

[DEP06] D. Duval, R. Echahed, and F. Prost. Modeling pointer redirection as cyclic term graph rewriting. In *TERMGRAPH 06*, 2006. Extended version to appear in ENTCS.

[DEP09] D. Duval, R. Echahed, and F. Prost. A heterogeneous pushout approach to term-graph transformation. In *Proceedings of Rewriting Techniques and Application 2009 (RTA'09)*, 2009.

[DHJ+06] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. Van Eetvelde. Adaptive star grammars. In *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2006.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics*, chapter 6, pages 243 – 320. Elsevier, Amsterdam, 1990.

[DP98] F. Damiani and F. Prost. Detecting and removing dead code using rank-2 intersection. In *International Workshop : TYPES'96, selected papers*, volume 1512 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[DP01] N. Dershowitz and D. A. Plaisted. Rewriting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes*, chapter 9, pages 535 –610. Elsevier and MIT Press, Amsterdam, 2001.

[EPR35] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality reality be considered complete ? *Phys. Rev.*, pages 777–780, 1935.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars : An algebraic approach. In *FOCS 1973*, pages 167–180, 1973.

[EPS03] R. Echahed, F. Prost, and W. Serwe. Statically assuring secrecy for dynamic concurrent processes. 2003. proceedings of PPDP'03, preliminary version avalaible at http ://www-leibniz.imag.fr/LesCahiers/2002/Cahier40/Resum-Cahier40.html.

[ES00] R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In John Lloyd et al., editors, *Proceedings of the* 1^{st} *International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 300 – 314, London, July 2000. Springer Verlag.

[ES02] R. Echahed and W. Serwe. Integrating action definitions into concurrent declarative programming. *Electronic Notes in Theoretical Computer Science*, 64, September 2002. special issue : selected papers of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001).

[Fey82] R. P. Feynmann. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7) :467–488, 1982.

[Fok00] Wan Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer Verlag, 2000.

[Gen09] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50 :1–102, 1987.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference. In *Proceedings of the* 31^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, Venice, Italy, January 2004.

[Gro96] L. K. Grover. A fast quantum mechanical algorithm for database search. In *28'th Symposium on the Theory Of Computing (STOC'96)*, pages 212–219. ACM, 1996.

[Hec77] M. C. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., 1977.

[HJ03] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, Proc. of the 30th ACM Symposium on Principles of Programming Languages (POPL 2003).

[Hoa69] T. Hoare. An axiomatic basis of computer programming. *CACM*, 12(10) :576–580, 1969.

[HR00] M. Hennessy and J. Riely. Information flow vs. ressource access in the asynchronous pi-calculus. In *Automata, Languages and Programming, 27th International Colloquium, (ICALP'2000), LNCS 1853*, pages 415–427. Springer, 2000.

[Hun91] S. Hunt. *Abstract Interpretation of functionnal languages : from theory to Practice.* PhD thesis, Department of Computing, Imperial College, London, 1991.

[JGS93] N.D. Jones, K. Gomard, and P Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, 1993.

[Kni96] E. Knill. Convention for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In B. Robinet, editor, *International Symposium on Programming, Proceedings of the Fourth 'Colloque International sur la Programmation', Paris, France, 22-24 April 1980*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer, 1980.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edimburgh, 1981.

[NC00] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principle of Program Analysis.* Springer, 1999.

[Per07] S. Perdrix. Quantum patterns and types for entanglement and separability. *Electronic Notes Theoretical Computer Science*, 170 :125–138, 2007.

[Per08] S. Perdrix. Quantum entanglement analysis based on abstract interpretation. In *Static Analysis, 15th International Symposium,*

SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings, volume 5079 of *Lecture Notes in Computer Science*, pages 270–282. Springer, 2008.

[PHW02a] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate confinement under uniform attacks. In M. V. Hermenegildo and G. Puebla, editors, *SAS'02 – Static Analysis, 9th International Symposium*, number 2477 in Lecture Notes in Computer Science, Madrid, Spain, September 2002. Springer.

[PHW02b] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *CSFW'02 – 15th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, 2002.

[Pie02] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Plu99] D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 3–61. World Scientific, 1999.

[Pow89] A. J. Power. An abstract formulation for rewrite systems. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science, Manchester, UK, September 5-8, 1989, Proceedings*, volume 389 of *Lecture Notes in Computer Science*, pages 300–312. Springer, 1989.

[Pro00] F. Prost. A static calculus of dependencies for the λ-cube. In *Proc. of IEEE 15th Ann. Symp. on Logic in Computer Science (LICS'2000)*. IEEE Computer Society Press, 2000.

[Pro01] F. Prost. On the semantics of non-interference type-based analyses. In *JFLA'001, Journées Francophones des Langages Applicatifs*, January 2001.

[PZ09] F. Prost and C. Zerrari. Reasoning about entanglement and separability in quantum higher-order functions. In *Proceedings of Unconventional Computation 2009 (UC'09)*, Lecture Notes in Computer Science. Springer, 2009. To be published.

[RMMG01] P.Y.A. Ryan, J. McLean, J. Millen, and V. Gilgor. Non-interference, who needs it ? In *CSFW'01 – 14th IEEE Computer Security Foundations Workshop*, pages 237 – 238, Cape Breton, Nova Scotia, Canada, June 2001.

[RS99] P.Y.A. Ryan and S.A. Schneider. Process algebra and non-interference. In *PCSFW : Proceedings of The 12th Computer*

Security Foundations Workshop. IEEE Computer Society Press, 1999.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[Sel06] P. Selinger. Special issue on quantum programming languages. *Mathematical Structures in Computer Science*, 16(3) :373–374, 2006.

[Sho94] P. W. Shor. Polynominal time algorithms for discrete logarithms and factoring on a quantum computer. In L. M. Adleman and M. A. Huang, editors, *Algorithmic Number Theory, First International Symposium, ANTS-I, Ithaca, NY, USA, May 6-9, 1994, Proceedings*, volume 877 of *Lecture Notes in Computer Science*. Springer, 1994.

[Sin00] S. Singh. *The Code Book : The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor, 2000.

[SM02] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, special issue on Design and Analysis Techniques for Security Assurance*, 2002. to appear.

[Smu68] R. M. Smullyan. *First-Order Logic*. Springer, 1968.

[SV98a] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 355–364. ACM, 1998.

[SV98b] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the* 25^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 355 – 364, San Diego, January 1998.

[SV05a] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *LNCS*, pages 354–368, 2005.

[SV05b] Peter Selinger and Benoit Valiron. A lambda calculus for quantum computation with classical control. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International*

Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings, volume 3461 of *LNCS*, pages 354–368, 2005.

[SVI96] Geoffrey Smith, Dennis M. Volpano, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3) :167 – 187, 1996.

[Vid03] G. Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters*, 91(14), 2003.

[VS98] D. Volpano and G. Smith. Confinements properties for programming languages. *SIGACT News*, 29(3) :33–42, 1998.

[Wei94] M. Weiser. Ubiquitous computing (abstract). In *ACM Conference on Computer Science*, page 418, 1994.

[Wei07] A. Weiss. Computing in the clouds. *netWorker*, 11(4) :16–25, 2007.

[YH00] N. Yoshida and M. Hennessy. Assigning types to processes. In IEEE Computer Society Press, editor, *Proc. of IEEE 15th Ann. Symp. on Logic in Computer Science (LICS'2000)*, pages 334–345, 2000.

[ZM01] S. Zdancewic and A. Myers. Robust declassification. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001., 2001.