Une méthode de sélection de tests à partir de spécifications algébriques.

Clément Boin

Thèse présentée par

Clément BOIN

pour obtenir le titre de

DOCTEUR de L'UNIVERSITÉ D'ÉVRY VAL D'ESSONNE

Spécialité : **Informatique**

Une méthode de sélection de tests à partir de spécifications algébriques

Date de soutenance : lundi 9 juillet 2007

Composition du jury

Yves BERTRAND	*Rapporteurs*
Régine LALEAU	
Marc AIGUIER	*Directeur de thèse*
Agnès ARNOULD	*Examinatrices*
Serena CERRITO	
Pascale LE GALL	

Thèse préparée au Laboratoire Informatique, Biologie Intégrative et Systèmes Complexes (IBISC) de l'Université d'Évry Val d'Essonne - FRE 2873 du CNRS.

Table des matières

Introduction

Lors de la réalisation d'un programme informatique, on distingue deux phases primordiales :

- la première phase, dite de $conception$, concerne les techniques utilisées pour construire ce programme. La conception comprend elle-même plusieurs étapes : il faut tout d'abord bien décrire les services attendus en fonction des $besoins$, ensuite il faut choisir la bonne $architecture$ (langage de programmation...), et enfin définir plus finement les modules, les fonctions, les structures de données, les algorithmes... Toutes ces étapes donnent lieu à des spécifications qui peuvent être de différentes natures (langage naturel, schémas, descriptions formelles...).
- la deuxième phase, dans laquelle s'inscrit cette thèse, consiste à $vérifier$ si le programme réalisé est conforme au projet de départ (c'est-à-dire aux spécifications).

Afin de développer des logiciels fiables ou répondant à des objectifs précis, il est nécessaire de réaliser ces deux étapes avec beaucoup de soin. C'est le but du *génie logiciel*.

Les *spécifications* décrivent les propriétés attendues du logiciel à différents niveaux d'abstractions et selon différents points de vue. Les spécifications sont un support non seulement pour la conception mais aussi pour la vérification du logiciel. En effet, elles constituent une référence de correction. La phase de vérification se fait par la mise en œuvre de techniques de test et/ou de preuve.

Les techniques de $preuve$ s'appuient sur des résultats théoriques, et permettent de prouver la validité de propriétés dans un programme. Mais leur mise en œuvre est un processus long et coûteux, notamment à cause de la multitude des interactions possibles avec l'environnement. En général, on prouve des petites parties d'un programme pour vérifier des propriétés dites de $sûreté$ (qui peuvent mettre la vie des personnes en danger par exemple). Citons la méthode B [Abr96] qui permet de le faire par raffinement successifs de la spécification. Cependant, la majeure partie des logiciels est vérifiée par d'autres méthodes dont le test fait partie.

Le $test$ est une méthode de vérification qui a pour but de détecter des fautes ou des inadéquations dans un logiciel [GMSB96, XRK00, Bei90]. C'est la méthode la plus communément utilisée pour assurer la qualité d'un logiciel. C'est une activité qui prend beaucoup de temps, et qui mérite que l'on s'intéresse à son développement. Les méthodes de test reposent sur le fait d'exécuter le logiciel sur un sous-ensemble fini bien choisi du domaine de ses entrées possibles (on parle de jeu de tests).

L'activité de test est réalisée en trois étapes principales : la *sélection* qui construit le jeu de test, la *soumission* qui consiste à confronter ce jeu de test à un programme, et enfin l'*oracle* qui permet de dire si un test est en succès ou en échec c'est-à-dire si le résultat du programme est conforme à celui attendu (donné par la spécification).

Il est possible de classer les méthodes de test en fonction du mode de sélection des jeux de tests qu'elles utilisent.

- Le *test structurel*, ou "boîte blanche" : la sélection des jeux de tests s'effectue à partir de la structure du code source représenté sous la forme d'un graphe. Les jeux de tests sont choisis selon différents critères de couverture du graphe structurel. Ce type de test bénéficie d'outils existants et est très utilisé pour le test de petites unités de programme. Dès que la taille de l'unité à tester augmente, la quantité de tests sélectionnés par les méthodes structurelles explose. De plus, il est inapte à détecter des chemins manquants (par exemple un oubli par rapport à la spécification), et à chaque nouvelle version du logiciel, il est nécessaire de relancer le processus de sélection.

- Le *test fonctionnel*, ou "boîte noire" : la sélection s'effectue à partir de la spécification, sans prendre en compte le code (on regarde ce que doit vérifier le programme mais on ne sait pas comment il le vérifie). Les jeux de tests sont sélectionnés selon des critères de couverture de la spécification. Comme les tests sont conçus indépendamment du code source, ils peuvent être réutilisés à plusieurs reprises, si la spécification ne change pas.

- Le *test statistique* : les tests sont choisis selon une répartition probabiliste parmi l'ensemble des données d'entrée possibles. Cette stratégie de sélection est largement répandue mais utilise le plus souvent des lois probabilistes simples. Cependant, une "bonne" couverture nécessite des lois probabilistes sophistiquées qui peuvent être très difficiles à construire. De plus, ces méthodes statistiques ne permettent pas d'adresser les données singulières (comme le traitement d'exception) qui ont une faible probabilité d'apparaître, mais qui jouent un rôle clé dans la sûreté des systèmes.

Toutes ces méthodes de sélection, si elles sont utilisées correctement et conjointement permettent de détecter la plupart des erreurs fréquentes ou graves. Ainsi le test est un bon moyen pour développer des logiciels fiables. Cependant, contrairement aux méthodes de preuve, le test n'offre pas de garantie de correction puisqu'il effectue une vérification partielle des propriétés.

Dans cette thèse nous nous intéressons au *test fonctionnel* et plus précisément à la définition de méthodes de sélection de tests à partir de spécifications algébriques. Notre approche s'inspire des travaux précédents [BGM91, Mar91b, Gau95, LGA96, Arn97, ALG02].

Cette thèse s'inscrit également dans la partie preuve, via la description d'une méthode de normalisation d'arbres de preuve. Celle-ci ne fait pas directement partie du domaine du test, mais de la démonstration automatique. Nous avons utilisé cette méthode de normalisation pour simplifier les preuves des propriétés de nos critères de sélection pour des spécifications algébriques.

Plan de la thèse

- La première partie de ce manuscrit contient un ensemble de rappels théoriques à propos des formalismes de spécifications et de la réécriture que nous utiliserons par la suite.
  - Dans le premier chapitre, nous présenterons la *logique des prédicats du premier ordre* en détail. Nous donnerons la syntaxe et la sémantique d'une version de cette logique où tous les objets du langage sont munis de sortes. Puis nous introduirons la notion générale de *systèmes formels*, qui nous permet de représenter les systèmes de preuve en logique : par exemple le calcul des séquents que nous présenterons pour la logique des prédicats du premier ordre. Nous aborderons ensuite une version particulière de la logique des prédicats du premier ordre restreinte au seul prédicat d'égalité : nous parlerons alors de *logique équationnelle*. Nous aborderons plus spécialement les formules conditionnelles positives qui sont les Clauses de Horn de la logique équationnelle. Puis pour terminer ce chapitre, nous parlerons des *spécifications formelles* qui sont des descriptions abstraites de programmes que l'on veut vérifier. Les formules équationnelles, notamment les formules conditionnelles positives, s'adaptent bien à ce type de descriptions. Nous donnerons plusieurs exemples de spécifications.
  - Dans le deuxième chapitre, nous nous intéresserons à la *réécriture*, et plus précisément à la réécriture de termes. Dans ce cadre, nous serons amenés à parler des notions d'unification et de terminaison qui sont fondamentales en informatique. Dans bien des cas, la terminaison ne peut

être prouvée qu'à l'aide d'ordres bien fondés particuliers, les *ordres récursifs sur les chemins* que nous présenterons en détail.

- La deuxième partie de cette thèse est consacrée à nos résultats de *normalisation d'arbres de preuve*. Ces résultats sont généraux parce qu'ils permettent de prendre en compte n'importe quel système formel (donc n'importe quel système de preuve) et de construire facilement des résultats de normalisation à l'aide de règles de transformation élémentaires d'arbres de preuve. Ces règles sont des règles de réécriture adaptées à la transformation d'arbres. Nous donnerons un résultat de normalisation forte, puis un résultat de normalisation faible que nous illustrerons à chaque fois par des exemples d'application (logicalité, lemme de Newman, élimination des coupures dans le calcul des séquents). Nous utiliserons largement ces résultats dans la dernière partie de cette thèse.
- La troisième et dernière partie est consacrée au problème de la sélection de tests à partir de spécifications algébriques qui est une version particulière du test fonctionnel. Elle est composée de trois chapitres.
  - Le premier propose un *état de l'art* sur le test fonctionnel. Nous serons amenés à présenter plusieurs outils d'aide à la sélection existants (HOL-TestGen, QuickCheck, LOFT) qui nous permettront de montrer l'importance du problème de la sélection des jeux de tests. Nous soulignerons ainsi l'intérêt de la sélection par partition en opposition à la sélection aléatoire.
  - Le deuxième chapitre est une présentation du *test fonctionnel* à partir de spécifications algébriques, c'est-à-dire notre cadre de travail. Nous verrons notamment comment sélectionner des tests à l'aide des critères d'uniformité et de régularité, nous généraliserons la notion de critère de sélection et enfin nous formaliserons des propriétés intéressantes pour les critères de sélection.
  - Puis, dans le troisième chapitre, nous présenterons nos deux critères de sélection de tests à partir de spécifications algébriques par *dépliage* des axiomes. Le premier critère est relatif à la sélection par dépliage de l'outil LOFT. Le second généralise le premier pour une plus large classe de spécifications. Nous garantirons les bonnes propriétés de ces deux critères en s'aidant notamment des résultats de normalisation d'arbres de preuve de la deuxième partie.

Première partie

Préliminaires : fondements théoriques

Chapitre 1

Formalismes de spécifications

Les objets et notions que nous allons utiliser tout au long de cette thèse sont liés à la logique [Lal90]. Elle joue un rôle fondamental en informatique, et notamment dans le domaine du test comme nous le verrons dans les derniers chapitres de ce manuscrit de thèse. En effet, lors de la conception d'un logiciel, avant l'étape de programmation, la première étape consiste à écrire une spécification de ce que l'on attend du logiciel. Celle-ci permet de décrire simplement ce que l'on attend du programme. Si cette spécification est écrite dans le langage courant des ambiguïtés sont possibles. L'écriture de cette spécification dans un formalisme logique permet d'éviter ce problème. De plus, ce langage formel est beaucoup plus proche de la syntaxe d'un langage de programmation. Il est donc possible de démontrer par des techniques mathématiquement fondées et vérifiables par ordinateur que des composants logiciels répondent logiquement à leur spécification (c'est ce qu'on appelle la correction formelle). Comme nous le verrons dans les prochains chapitres de cette thèse, un tel langage permet également dans le cadre du test fonctionnel de tester (plus facilement) si le programme correspond à sa spécification. Mais avant ceci, nous allons présenter le cadre et les différentes techniques qui vont être par la suite utiles à notre propos.

Dans un premier temps, nous présenterons la logique des prédicats du premier ordre qui est le cadre théorique des spécifications que nous avons choisi. Nous la présenterons classiquement sous les trois aspects des langages formels : la syntaxe, la sémantique, et les preuves. Nous étudierons un système de preuve très utilisé, notamment en déduction automatique : le calcul des séquents. De plus, la logique des prédicats du premier ordre est très utilisée dans le domaine des spécifications de logiciels car elle possède de nombreuses propriétés algébriques et de décidabilité.

Nous parlerons ensuite d'une classe particulière de la logique des prédicats du premier ordre : la logique équationnelle. La notion d'équation est importante dans le cadre du test fonctionnel car elle permet de spécifier l'application d'une

fonction sur des arguments ainsi que le résultat attendu de cette application.

Dans le cadre de la logique équationnelle, nous étudierons plus particulièrement un système de spécifications, dites algébriques, dont les formules sont une restriction des clauses de Horn réduites aux équations (ces formules viennent du langage de programmation PROLOG).

1.1 La logique des prédicats du premier ordre

Nous allons présenter la logique des prédicats du premier ordre en trois étapes importantes. Les deux premières sont données dans cette section, et la troisième est présentée séparément dans la section suivante. La première étape consiste à définir la *syntaxe* de notre langage, c'est-à-dire quelles sont les formules que nous allons considérer. La deuxième étape est la définition de la *sémantique*, c'est-à-dire le sens mathématique que l'on donne aux symboles de la syntaxe. Puis pour finir, nous définirons un *système de preuve* qui permet de raisonner mécaniquement sur l'ensemble des formules, c'est-à-dire d'une manière compréhensible par un ordinateur.

1.1.1 Syntaxe

La syntaxe permet de spécifier quelles expressions du langage sont des formules de notre logique. La syntaxe est définie indépendamment de la signification et de la valeur de vérité des symboles. La syntaxe d'un formalisme logique se définit toujours à l'aide des notions de signature et de formules construites sur une signature.

1.1.1.1 Signature

La signature d'un langage du premier ordre contient l'ensemble des symboles de fonctions et de prédicats qui vont être utiles à la spécification du problème visé. Ainsi la signature peut être vue comme "l'interface" du logiciel à spécifier. Nous considérons ici une version multi-typée de la logique des prédicats du premier ordre, c'est-à-dire une version où tous les objets et symboles du langage sont munis d'un type.

Définition 1.1.1 (Signature) *Une* **signature** *du premier ordre est un quadruplet* (S, F, R, V) *où* S *est un ensemble dont les éléments sont appelés* **sortes** *(ou* **types***),* F *et* R *sont des ensembles disjoints tels que :*

– *F est un ensemble de symboles d'opération (ou de fonction), où chaque symbole est muni d'un profil* [1] *dans S^+,*
– *R est un ensemble de symboles de prédicats, où chaque symbole est muni d'un profil dans S^+,*

et V une famille indexée par S d'ensembles V_s telle que pour chaque $s \in S, V_s \cap F = \emptyset$. Pour chaque $s \in S$, les éléments de V_s sont appelés **variables de sorte** s.

On notera $f : s_1 \times \cdots \times s_n \rightarrow s$ un symbole de fonction muni du profil $s_1 \ldots s_{n+1} \in S^+$ avec $s_1, \ldots, s_n$ les sortes des arguments, et s_{n+1} la sorte du résultat, et $p : s_1 \times \cdots \times s_n$ un symbole de prédicat muni du profil $s_1 \ldots s_n \in S^+$.

Si n vaut 0 pour un symbole de fonction f alors f est appelé **symbole de constante** *de sorte s, et on le note $f :\rightarrow s$ (ou bien $f : s$).*

Quand il n'y a pas d'ambiguïté (en fonction du contexte) sur le profil d'un symbole de fonction ou de prédicat, on omettra de l'indiquer à chaque fois. Intuitivement, les symboles de prédicats sont des fonctions qui ont une valeur booléenne (vraie ou fausse) dépendante des arguments. Dans la suite, on parlera souvent du symbole d'égalité $=$ qui est un symbole de prédicat.

Exemple 1.1.1 *Par exemple si l'on veut exprimer la propriété suivante $(x > 0) \wedge (x + 0 = x)$, nous avons besoin des symboles $0, +, >, =$.*
Un seul type de données est nécessaire pour construire une telle expression. Nous pouvons, par exemple, utiliser les entiers naturels que l'on désignera à l'aide du nom de sorte $entier$. La signature devra donc au minimum être composée :

– *d'un symbole de constante $0 : entier$,*
– *d'un symbole $+ : entier \times entier \rightarrow entier$,*
– *des deux symboles de prédicats $>, =: entier * entier$,*
– *d'une variable x de sorte $entier$.*

Le connecteur logique $\wedge$ et les parenthèses ne font pas partie de la signature car ce sont des symboles du langage du premier ordre et ils ne changent pas quelque soit la signature. Nous le verrons lorsque nous définirons la notion de formule.

Nous pouvons enrichir cette signature à l'aide des symboles de fonction $succ : entier \rightarrow entier$ et $* : entier \times entier \rightarrow entier$ qui représentent respectivement le " $+ 1$" ($succ$ est une abréviation de successeur) et la multiplication sur les entiers. Nous pouvons également y ajouter des variables.

[1] Soit S un ensemble. S^+ désigne l'ensemble de tous les mots non vides finis définis sur l'alphabet S. Pour une définition rigoureuse de S^+ se reporter à la définition 1.2.1 de ce chapitre.

Nous obtenons ainsi la signature $\Sigma_{Arith} = (S_{Arith}, F_{Arith}, R_{Arith}, V_{Arith})$ *de l'arithmétique usuelle telle que :*

$$\begin{aligned} S_{Arith} &= \{entier\} \\ F_{Arith} &= \{0 : entier, succ : entier \rightarrow entier, \\ &\quad + : entier \times entier \rightarrow entier, \\ &\quad * : entier \times entier \rightarrow entier\} \\ R_{Arith} &= \{=: entier \times entier, >: entier \times entier\} \\ V_{Arith} &= \{x, y, z : entier\} \end{aligned}$$

Remarque 1.1.1 *Dans la suite, tout symbole de prédicat binaire (c'est-à-dire dont le profil est de la forme* $s_1 \times s_2$*) sera noté sous sa forme infixe* [2] *plutôt que sous la forme préfixe* [3] *utilisée pour la notation fonctionnelle. Certains symboles de fonctions comme le* $+$ *de l'arithmétique le seront également. Dans la signature, pour tout symbole* @ *utilisé de manière infixe, on notera* _@_*.*

1.1.1.2 Termes

Lorsque la signature est connue, elle nous donne tous les "ingrédients" permettant de construire les formules bien formées de la logique considérée. Dans la logique des prédicats du premier ordre les formules vont exprimer des propriétés à l'aide de prédicats (notamment l'égalité) et de connecteurs et quantificateurs logiques. Ces propriétés s'expriment sur les valeurs d'un ou plusieurs types. Ces valeurs doivent donc aussi être représentées de façon symbolique à partir des éléments introduits dans la signature. Ces valeurs sont communément appelées *termes avec variables*. Ils sont construits par composition des noms de fonctions de la signature et des variables. Les termes doivent être construits en respectant les sortes données dans la signature. Il faut respecter le profil de symboles de fonctions et le type des variables. Les variables permettent de manipuler des valeurs génériques, c'est-à-dire une multitude (souvent infinie) de valeurs définies par une forme commune. Formellement les termes se définissent de la façon suivante :

Définition 1.1.2 (Termes) *Soit* $\Sigma = (S, F, R, V)$ *une signature. Pour tout* $s \in S$*, on définit* **l'ensemble $T_\Sigma(V)_s$ des termes avec variables de type** s*, comme le plus petit ensemble au sens de l'inclusion satisfaisant les conditions suivantes :*

- V_s *est inclus dans* $T_\Sigma(V)_s$ *;*

[2] Infixe : notation pour laquelle les deux arguments d'un symbole binaire sont placés à gauche et à droite de ce symbole.

[3] Préfixe : les arguments se trouvent après le symbole binaire considéré.

– *pour tout symbole de fonction* $f : s_1 \times \cdots \times s_n \to s$ *de* F *et tout* n*-uplet de termes* $(t_1, \ldots, t_n)$ *de* $T_\Sigma(V)_{s_1} \times \cdots \times T_\Sigma(V)_{s_n}$, $f(t_1, \ldots, t_n)$ *est un terme de* $T_\Sigma(V)_s$. *Si* f *est une constante, on note* f *à la place de* $f()$.

On note $T_\Sigma(V) = (T_\Sigma(V)_s)_{s \in S}$ *la famille d'ensembles des* **termes avec variables**.

Dans la suite, on notera $Var(t)$ *l'ensemble des variables apparaissant dans un terme* t, *et* $t : s$ *un terme de sorte* s. *Lorsque* $Var(t)$ *est vide, on dit que* $t : s$ *est un* **terme clos de type** s *et l'ensemble des termes clos de type* s *est noté* $T_{\Sigma s}$.

Deux signatures différentes engendreront deux ensembles de termes différents.

Par convention, on notera souvent les symboles de fonctions à l'aide des lettres f, g, h, les symboles de prédicats à l'aide des lettres p, q, r et les variables à l'aide des lettres x, y, z.

Exemple 1.1.2 *Soit* $\Sigma_{Arith} = (S_{Arith}, F_{Arith}, R_{Arith}, V_{Arith})$ *la signature de l'arithmétique vue précédemment.* $succ(x)+succ(succ(0))$ *et un terme de* $T_{\Sigma_{Arith}}(V)_{entier}$.

1.1.1.3 Formules

À partir des termes avec variables, nous pouvons maintenant énoncer l'objet de notre syntaxe qui est la définition des formules de la logique du premier ordre. Les formules permettent d'exprimer des propriétés sur des termes à l'aide des symboles de prédicats et des connecteurs logique. Nous définissons tout d'abord une notion intermédiaire où l'on se restreint aux symboles de prédicat : les formules atomiques.

Définition 1.1.3 (Formules atomiques) *Soit* $\Sigma = (S, F, R, V)$ *une signature, une* **formule atomique** *est une expression de la forme* $p(t_1, \ldots, t_n)$ *où* p *est un symbole de prédicat de profil* $s_1 \times \cdots \times s_n$ *et* $(t_1, \ldots, t_n)$ *un n-uplet de termes de* $T_\Sigma(V)_{s_1} \times \cdots \times T_\Sigma(V)_{s_n}$.

Exemple 1.1.3 *Les expressions* $(x > 0)$ *et* $(x + 0 = x)$ *sont des formules atomiques car* 0, x *et* $x + 0$ *sont des termes de type* $entier$.

Par convention, on notera souvent les formules avec des lettres majuscules de l'alphabet grec : $\Phi, \Phi', \Psi, \ldots$

Les formules sont définies de façon inductive à partir de l'ensemble de base des formules dites atomiques. Elles sont enrichies par des connecteurs logique et par deux quantificateurs.

Définition 1.1.4 (Formule) *Soit $\Sigma = (S, F, R, V)$ une signature. L'ensemble de* **formules** *$For(\Sigma)$ est le plus petit ensemble (au sens de l'inclusion) qui satisfait les conditions suivantes :*

- *il contient toutes les* formules atomiques,
- *à chaque fois qu'il contient une formule Φ, il contient également :*

$$(\neg\Phi)$$

- *à chaque fois qu'il contient deux formules Φ et Ψ, il contient également :*

$$(\Phi \wedge \Psi) \quad , \quad (\Phi \vee \Psi) \quad , \quad (\Phi \Rightarrow \Psi)$$

- *à chaque fois qu'il contient une formule Φ, il contient pour toute variable $x \in V$ de sorte $s \in S$:*

$$(\forall x.\Phi) \quad , \quad (\exists x.\Phi)$$

Les symboles $\wedge, \vee, \Rightarrow, \neg$ sont les **connecteurs logique**, *$\forall$ est le* **quantificateur universel**, *et $\exists$ et le* **quantificateur existentiel**.

Pour réduire le nombre de parenthèses dans une formule, on impose les conventions suivantes :

- on peut éliminer les parenthèses $(\ )$ les plus extérieures,
- $\wedge$ et $\vee$ sont prioritaires par rapport à $\Rightarrow$ (*exemple :* $A \Rightarrow B \vee C$ se lit $A \Rightarrow (B \vee C)$),
- $\neg$ s'applique à la plus petite formule suivante (*exemple :* $\neg p \wedge q$ se lit $(\neg p) \wedge q$),
- si on écrit $A * B * C * D$ (où $*$ représente toujours le même connecteur) on lira $(((A * B) * C) * D)$ (*association à gauche*).

Exemple 1.1.4 *Notre expression $(x > 0) \wedge (x + 0 = x)$ est donc bien une formule du premier ordre. Nous pourrions aussi l'écrire de la manière suivante : $\forall x : entier.(x > 0) \wedge (x + 0 = x)$. On dit alors que les variables sont quantifiées universellement par la variable x. Parfois, on omettra d'indiquer le signe $\forall$, on dira alors que les variables sont implicitement quantifiées universellement.*

Pour alléger la notation nous avons omis d'indiquer certaines parenthèses. Pour être totalement conforme avec la définition de formule nous aurions dû écrire : $(\forall x.((> (x, 0)) \wedge (= (+(x, 0), x))))$

1.1.2 Sémantique de la logique des prédicats du premier ordre

Le but de la sémantique est de donner un sens mathématique aux constructions syntaxiques de la logique. Pour cela, il faut donner une signification aux symboles de la signature et une valeur aux variables. Il faut également donner la signification des connecteurs et des quantificateurs.

Exemple 1.1.5 *Intuitivement, la formule équationnelle*

$$succ(succ(0)) + succ(0) = succ(succ(succ(0)))$$

signifie $2 + 1 = 3$ *dans l'ensemble des entiers naturels* $\mathbb{N}$. *Cette équation entre deux entiers naturels est évidemment vraie.*

1.1.2.1 Σ-Modèles

Commençons par donner un sens mathématique aux signatures à l'aide de la notion de modèle. Il est décrit simplement au moyen d'un *ensemble* muni d'*opérations internes* et de *relations*. Dans le cas de la logique munie de sortes cet ensemble est en fait une famille d'ensembles indexées par les sortes de la signature, les S-ensembles :

Définition 1.1.5 (S-ensemble) *Soit* S *un ensemble de sortes. Un* S-**ensemble** M *est une famille* [4] *d'ensembles indexée par* S, *c'est-à-dire* $M = (M_s)_{s \in S}$.

Exemple 1.1.6 *Pour un ensemble de sortes* $S = \{entier, bool, listes\}$, *un* S-*ensemble est composé des trois sous-ensembles* $M_{entier}, M_{bool}, M_{listes}$.

Le S-*ensemble* pour un modèle donné caractérisera le domaine des individus. Les *opérations internes* donneront une signification mathématique (ou sémantique) aux symboles de fonctions et les *relations* une signification aux symboles de prédicats de la signature.

Définition 1.1.6 (Σ-Modèle) *Soit* $\Sigma = (S, F, R, V)$ *une signature. Un* Σ-**modèle** $\mathcal{M}$ *est un* S-*ensemble* M *muni :*

- *pour chaque symbole de fonction* $f : s_1 \times \cdots \times s_n \to s$, *d'une application* $f^{\mathcal{M}} : M_{s_1} \times \cdots \times M_{s_n} \to M_s$ *(si* $n = 0$ *alors* $f^{\mathcal{M}}$ *est un élément de* M_s*),*
- *pour chaque symbole de prédicat* $p : s_1 \times \cdots \times s_n$, *d'un sous-ensemble* $p^{\mathcal{M}}$ *de* $M_{s_1} \times \cdots \times M_{s_n}$.

On note $Mod(\Sigma)$ *la classe des* Σ-*modèles.*

[4] Une famille d'ensembles est simplement un ensemble d'ensembles

Exemple 1.1.7 *Pour l'exemple de la signature Σ_{Arith}, on peut considérer le modèle $\mathcal{M}_{Arith}$ de l'arithmétique usuelle, composé :*

- *de l'ensemble $M_{entier} = \mathbb{N}$ des entiers naturels*
- *de la fonction $0^{\mathcal{M}_{Arith}} = 0_{\mathbb{N}}$,*
- *de la fonction $succ^{\mathcal{M}_{Arith}} = \_ +_{\mathbb{N}} 1_{\mathbb{N}}$ qui ajoute 1 à son argument noté $\_$,*
- *des fonctions binaires $+^{\mathcal{M}_{Arith}}$ et $\times^{\mathcal{M}_{Arith}}$, qui sont respectivement l'addition $+_{\mathbb{N}}$ et la multiplication $\times_{\mathbb{N}}$ dans les entiers,*
- *d'un sous-ensemble $=^{\mathcal{M}_{Arith}}$ des couples $\mathbb{N} \times \mathbb{N}$ qui sont égaux entre eux, et d'un ensemble $>^{\mathcal{M}_{Arith}}$ des couples d'entiers dont le premier est supérieur au second.*

Dans cet exemple, le Σ-modèle construit correspond au comportement habituel des fonctions arithmétiques mais nous pourrions tout aussi bien construire un modèle qui associe à ces symboles de fonctions un comportement valide tout à fait différent.

1.1.2.2 Évaluation des termes

À partir d'un Σ-modèle, on voit bien comment donner un sens mathématique aux termes en appliquant la stratégie qui consiste d'abord à évaluer les arguments des fonctions avant d'appliquer la fonction dont le sens est donné par le Σ-modèle. Le seul problème est que les Σ-modèles ne permettent pas d'interpréter les variables, donc de donner la valeur de vérité d'une formule. On commence alors à donner un sens aux variables.

Définition 1.1.7 (Interprétation) *Soient $\Sigma = (S, F, R, V)$ une signature et $\mathcal{M}$ un Σ-modèle. Une* **interprétation**, *notée ι, est une famille indexée par S d'applications $\iota_s : V_s \to M_s$.*

Ceci revient à associer à chaque variable une valeur dans le modèle $\mathcal{M}$, de même sorte que la variable. On peut étendre naturellement la notion d'interprétation aux termes avec variables et ainsi donner un sens mathématique aux termes avec variables. Ceci se fait à l'aide de l'application $\iota^\sharp : T_\Sigma(V) \to M$ définie inductivement sur la structure des termes avec variables de $T_\Sigma(V)$ par $x \mapsto \iota(x)$ et $f(t_1, \ldots, t_n) \mapsto f^{\mathcal{M}}(\iota^\sharp(t_1), \ldots, \iota^\sharp(t_n))$.

1.1.2.3 Relation de satisfaction

Grâce aux notions de Σ-modèle et d'interprétation définies précédemment, il est désormais possible de savoir si une formule de $For(\Sigma)$ est satisfaite ou non par

un modèle. Il faut pour cela étendre toute interprétation $\iota : V \to M$ en un prédicat sur les formules $\mathcal{M} \models^{\iota}$. Ceci se définit formellement de la manière suivante :

Définition 1.1.8 (Satisfaction) *Soient $\Sigma = (S, F, R, V)$ une signature, $\mathcal{M}$ un Σ-modèle et $\varphi \in For(\Sigma)$ une formule. $\mathcal{M}$* **satisfait** *la formule φ pour une interprétation $\iota : V \to M$, noté $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$, si et seulement si :*

- *si φ est une formule atomique de la forme $p(t_1, \ldots, t_n)$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$ si et seulement si $(\iota^{\sharp}(t_1), \ldots, \iota^{\sharp}(t_n)) \in p^{\mathcal{M}}$,*
- *si φ est de la forme $\neg\psi$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$ ssi $\mathcal{M} \not\models^{\iota}_{\Sigma} \psi$,*
- *si φ est de la forme $\psi_1 \wedge \psi_2$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$ ssi $\mathcal{M} \models^{\iota}_{\Sigma} \psi_1$ et $\mathcal{M} \models^{\iota}_{\Sigma} \psi_2$,*
- *si φ est de la forme $\psi_1 \vee \psi_2$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$ ssi $\mathcal{M} \models^{\iota}_{\Sigma} \psi_1$ ou $\mathcal{M} \models^{\iota}_{\Sigma} \psi_2$,*
- *si φ est de la forme $\psi_1 \Rightarrow \psi_2$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$ ssi si $\mathcal{M} \models^{\iota}_{\Sigma} \psi_1$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \psi_2$,*
- *si φ est de la forme $\exists x.\psi$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$ ssi il existe une interprétation ι' identique à ι sauf éventuellement pour la variable x, telle que $\mathcal{M} \models^{\iota'}_{\Sigma} \psi$,*
- *si φ est de la forme $\forall x.\psi$ alors $\mathcal{M} \models^{\iota}_{\Sigma} \varphi$ ssi pour toute interprétation ι' identique à ι sauf éventuellement pour la variable x, $\mathcal{M} \models^{\iota'}_{\Sigma} \psi$,*

La vérification d'une propriété φ pour un Σ-modèle $\mathcal{M}$ va donc consister à vérifier la satisfaction de φ pour toutes les interprétations de variables libres (c'est-à-dire qui ne sont pas dans la portée d'un quantificateur). Les variables dénotent ainsi des valeurs génériques. On parle alors de validation de formules.

Définition 1.1.9 (Validation) *Un Σ-modèle $\mathcal{M}$* **valide** *φ, noté $\mathcal{M} \models_{\Sigma} \varphi$, si et seulement si pour toute interprétation $\iota : V \to M, \mathcal{M} \models^{\iota}_{\Sigma} \varphi$.*

Lorsque qu'il n'y a pas d'ambiguïté sur la signature Σ considérée, on note $\models^{\iota}$ à la place de $\models^{\iota}_{\Sigma}$, et $\models$ à la place de $\models_{\Sigma}$.

Exemple 1.1.8 *Le modèle $\mathcal{M}_{Arith}$ de l'arithmétique usuelle satisfait la formule $succ(succ(0)) = x$ pour l'interprétation $\iota : x \mapsto 2$, mais ne la satisfait pas pour toutes les autres interprétations.*

Le modèle $\mathcal{M}_{Arith}$ valide la formule $succ(succ(x)) = x + succ(succ(0))$ car le modèle satisfait cette formule quelque soit l'interprétation de x.

Il est possible de caractériser des modèles équivalents pour un ensemble de formules Ψ. Cela signifie que ces deux modèles valident les mêmes formules de Ψ.

Définition 1.1.10 *Soit Σ une signature, et $\Psi \in For(\Sigma)$ un ensemble de Σ-formules.*

Deux modèles $\mathcal{M} \in Mod(\Sigma)$ et $\mathcal{N} \in Mod(\Sigma)$ sont **Ψ-équivalents**, *noté $\mathcal{M} \equiv_{\Psi} \mathcal{N}$, si et seulement si ils valident les mêmes formules de Ψ, c'est-à-dire :*

$$(\mathcal{M} \equiv_{\Psi} \mathcal{N}) \Leftrightarrow (\forall \varphi \in \Psi, \mathcal{M} \models \varphi \Leftrightarrow \mathcal{N} \models \varphi)$$

1.2 Les systèmes de preuve

1.2.1 Introduction

La sémantique présentée dans la section précédente permet de raisonner de façon rigoureuse sur des objets logique. Il est ainsi possible de garantir la correction de propriétés dans la logique considérée. Le problème de ce type de raisonnement est que nous nous permettons toute la puissance des mathématiques ainsi que des détours classiquement utilisées dans ce type de preuve comme “il est connu que”, “découle directement de”, etc. Or, dans le cadre de l’informatique et plus précisément du génie logiciel, on ne peut pas se contenter de tels détours. Nous devons certifier la qualité des preuves, c’est-à-dire vérifier mécaniquement qu’aucune erreur de raisonnement ne s’est glissée dans la preuve. La question qui se pose est de trouver comment aboutir à démontrer qu’une suite de symboles (une formule en l’occurrence) est correcte. La seule chose que sache faire un ordinateur étant de manipuler des symboles, il faut alors définir les conditions de manipulation de ces formules pour obtenir une formule correcte. On parle alors de systèmes de preuve, systèmes d’inférence, ou encore de calcul.

Pour les formules de la logique des prédicats du premier ordre, il existe de nombreux systèmes de preuve différents mais tous équivalents en pouvoir de preuve : la déduction naturelle, le calcul des séquents, les tableaux... Nous allons présenter ici le calcul des séquents. Le calcul des séquents est un système, introduit par le logicien allemand Gentzen [Gen35], qui permet de construire la preuve d’une formule sans ajouter aucun élément extérieur, et donc très pratique pour la déduction automatique. Il existe de nombreuses variantes : nous allons étudier ici une version du calcul des séquents (inspirée du système de Gentzen **LK** pour la logique classique) que nous utiliserons largement dans la deuxième partie de cette thèse, notamment pour montrer un des résultats les plus fondamentaux de la logique : l’élimination des coupures.

Avant de présenter le calcul des séquents nous allons étudier un cadre générique très bien adapté à une représentation générale des systèmes de preuve en logique : les systèmes formels. Ces derniers seront à la base des résultats établis dans la deuxième partie.

1.2.2 Les systèmes formels

Les systèmes formels sont des constructions syntaxiques rigoureuses qui permettent de représenter des mécanismes de production d’énoncés. Ils permettent, par exemple, de représenter tous les systèmes de preuves que nous allons aborder dans cette thèse.

1.2.2.1 Les mots

Les mots sont les objets élémentaires du paysage syntaxique de l'informatique. Ils sont connus sous le nom de chaînes de caractères. Ils permettent de construire tous les objets plus complexes que nous manipulerons dans la suite.

Définition 1.2.1 (Alphabet) *Soit A un ensemble, appelé* **alphabet**, *et dont les éléments sont appelés* **symboles**. *Un* **mot** *sur A est une suite finie d'éléments de A. On note A^* l'ensemble de tous les mots possibles, et A^n, avec $n \in \mathbb{N}$, l'ensemble de tous les mots de longueur n. On note A^+ l'ensemble de tous les mots A^n sachant que $n > 0$.*

A^* est muni d'une opération binaire, la concaténation, notée par la juxtaposition de deux mots (on la notera également à l'aide d'un point .) :

$$\begin{array}{rcl} A^p \times A^q & \rightarrow & A^{p+q} \\ (u, v) & \mapsto & uv \end{array}$$

avec $uv = (u_1, \ldots, u_p, v_1, \ldots, v_q)$ si $u = (u_1, \ldots, u_p)$ et $v = (v_1, \ldots, v_q)$.

Cette opération est associative et a pour élément neutre le mot vide ε qui est l'unique élément de A^0. Ainsi, $(A^*, ., \varepsilon)$ est un monoïde.

Exemple 1.2.1 *Soit L un ensemble composé de toutes les lettres de l'alphabet latin $\{a, b, c, \ldots, z\}$.*
$''exemple''$ est un mot de longueur 7 construit sur l'alphabet L. Il fait partie de l'ensemble L^7 qui est un sous-ensemble de L^*.

Un alphabet permet de représenter tous les mots dont on pourrait avoir besoin mais ne nous donne aucune indication sur la manière de construire des mots "corrects". Rien ne nous empêche dans l'exemple précédent de construire des mots qui ne sont pas dans le dictionnaire. Il nous faut alors un ensemble de règles de production d'énoncés corrects. Un alphabet auquel on ajoute des règles de production forme ce que l'on appelle un système formel.

1.2.2.2 Les systèmes formels

Voici la définition générale de la notion de système formel. Celle-ci nous donne un cadre abstrait que nous spécialiserons en fonction des besoins dans la suite du manuscrit.

Définition 1.2.2 (Système formel) *Un* **système formel** *(ou* **calcul***) $\mathcal{S}$ est défini par un triplet $(\mathcal{A}, \mathcal{F}, \mathcal{R})$ tel que :*

- *$\mathcal{A}$ est un* **alphabet** *;*
- *$\mathcal{F}$ est un sous-ensemble de $\mathcal{A}^*$, dont les éléments sont appelés* **formules bien formées** *;*

– *$\mathcal{R}$ est un ensemble fini de relations n-aires sur les formules ; chaque relation est appelée* **règle d'inférence**.

En pratique, l'ensemble $\mathcal{F}$ n'est pas quelconque, il est défini inductivement par une grammaire ou tout autre algorithme de reconnaissance de formules bien formées. Par exemple à l'aide des formules de la logique des prédicats du premier ordre présentées section 1.1.1.3 page 17. Cet ensemble diffère d'un système formel à l'autre.

Notation 1.2.1 *Soit $\mathcal{S}$ un système formel. Pour tout $r \in \mathcal{R}$, si $(\varphi_1, \ldots, \varphi_n) \in r$, avec $n \geq 1$, alors les formules $\varphi_1, \ldots, \varphi_{n-1}$ sont appelées* **prémisses** *et φ_n* **conclusion** *de l'instance.*
On notera une instance de cette règle d'inférence r par :

$$\frac{\varphi_1, \ldots, \varphi_{n-1}}{\varphi_n}\ r$$

Les instances de règles d'arité 1 sont appelées **axiomes** *et sont notées $\frac{}{\varphi_1}$.*

1.2.2.3 Un exemple simple

À titre d'exemple, voici le système formel $\mathcal{S}_{\alpha\beta} = (\mathcal{A}_{\alpha\beta}, \mathcal{F}_{\alpha\beta}, \mathcal{R}_{\alpha\beta})$ qui représente l'ordre alphabétique :

– $\mathcal{A}_{\alpha\beta} = L \cup \{<\}$ où L est l'ensemble des vingt-six lettres de l'alphabet latin ;
– $\mathcal{F}_{\alpha\beta} = \{\alpha < \beta | \alpha, \beta \in L^*\}$;
– $\mathcal{R}_{\alpha\beta}$ contient les schémas de règles d'inférence suivants :
  – $\frac{}{\varepsilon < x}\ ax1_x$ pour chaque x de L, où ε est le mot vide ;
  – $\frac{}{x < y}\ ax2_{xy}$ pour tout couple (x, y) de $L^1 \times L^1$, si x est avant y dans l'alphabet ;
  – pour tout couple (Φ, Ψ) de $\mathcal{F}$, et toute lettre x de L, on a les trois règles suivantes :

$$\frac{\Phi < \Psi}{x\Phi < x\Psi}\ gd_x$$

$$\frac{\Phi < \Psi}{\Phi < \Psi x}\ d_x$$

$$\frac{\Phi < \Psi}{\Phi x < \Psi}\ g_x \quad \text{si } \Phi \text{ est de longueur supérieure à celle de } \Psi$$

1.2.2.4 Déduction

À partir d'un ensemble d'hypothèses composé de formules et par application d'un ensemble de règles d'inférences, il est possible de calculer d'autres formules qui en sont déduites. On appelle *formules inférées* les formules atteintes, et *déduction* la trace des formules intermédiaires utilisées pour atteindre les formules inférées. Ces mécanismes de déduction sont les mêmes quelque soit le système formel considéré.

Définition 1.2.3 (Déduction) *Soit $\mathcal{S} = (\mathcal{A}, \mathcal{F}, \mathcal{R})$ un système formel et $\Gamma \subseteq \mathcal{F}$ un ensemble de formules. Une* **déduction** *dans $\mathcal{S}$ à partir de l'ensemble d'hypothèses Γ est une séquence $(\varphi_1, \ldots, \varphi_n)$ de formules telle que, pour tout $i \in \{1, \ldots, n\}$,*

- *ou bien φ_i appartient à Γ,*
- *ou bien il existe une instance $\frac{\psi_1, \ldots, \psi_k}{\varphi_i}$ d'une règle de $\mathcal{R}$ telle que le multi-ensemble*[5] *$\{\{\psi_1, \ldots, \psi_k\}\}$ est inclus dans le multi-ensemble $\{\{\varphi_1, \ldots, \varphi_{i-1}\}\}$.*

La dernière formule d'une déduction à partir d'un ensemble Γ est dite *inférée* à partir de Γ. Nous pouvons en déduire une relation binaire $\vdash_{\mathcal{S}}$ entre les ensembles de formules (les hypothèses) et les formules (celles qui sont déduites), qui est appelée *relation d'inférence*.

Définition 1.2.4 (Relation d'inférence) *Soient $\mathcal{S} = (\mathcal{A}, \mathcal{F}, \mathcal{R})$ un système formel, $\Gamma \subseteq \mathcal{F}$ un ensemble de formules et $\varphi \in \mathcal{F}$ une formule. On dit que Γ* **infère** *φ dans $\mathcal{S}$, et on écrit $\Gamma \vdash_{\mathcal{S}} \varphi$, si et seulement s'il existe une déduction dans $\mathcal{S}$ à partir de Γ dont la dernière formule est φ. Quand il n'y a pas d'ambiguïté sur le système formel utilisé, celui-ci est omis et on note $\Gamma \vdash \varphi$.*

1.2.2.5 Arbres de preuve

Pour obtenir les énoncés à l'aide des règles d'inférence, nous pouvons aussi utiliser un schéma d'induction ayant pour *base* un ensemble d'hypothèses et les axiomes du système formel, et pour *étape d'induction* les règles d'inférence du système formel. On parle alors plutôt de théorèmes.

Définition 1.2.5 (Théorèmes) *Soient $\mathcal{S} = (\mathcal{A}, \mathcal{F}, \mathcal{R})$ un système formel et Γ un ensemble de formules de $\mathcal{F}$. L'ensemble des* **théorèmes** *déduits de Γ dans $\mathcal{S}$ est le sous-ensemble $Th_{\mathcal{S}}(\Gamma)$ de $\mathcal{F}$ défini selon le schéma d'induction suivant :*

- base *: les axiomes de $\mathcal{R}$ et les formules de Γ sont des théorèmes ;*

[5] Un *multi-ensemble* a les mêmes propriétés qu'un ensemble mais accepte plusieurs occurrences d'un même élément (voir section 2.4.1.3 page 53 pour une description détaillée)

– induction *: si toutes les prémisses d'une instance d'une règle de $\mathcal{R}$ sont des théorèmes, alors sa conclusion est aussi un théorème.*

Lorsque Γ est l'ensemble vide, les formules obtenues sont appelées **tautologies** *et elles forment l'ensemble $Th_{\mathcal{S}}(\emptyset)$.*

On notera $Th(\Gamma)$ lorsqu'aucune ambiguïté n'est possible sur $\mathcal{S}$.

L'ensemble des formules pour lesquelles il existe une déduction et celui des théorèmes ne font qu'un, comme l'établit le résultat suivant :

Proposition 1.2.1 *Soient $\mathcal{S} = (\mathcal{A}, \mathcal{F}, \mathcal{R})$ un système formel et Γ un ensemble de formules de $\mathcal{F}$. On a l'équivalence suivante :*

$$\forall \varphi \in \mathcal{F},\ \ \Gamma \vdash_{\mathcal{S}} \varphi \ \Leftrightarrow\ \varphi \in Th_{\mathcal{S}}(\Gamma)$$

Preuve

$\Rightarrow$ Ce sens se prouve par récurrence sur la longueur de la déduction de φ.

– Cas de base : Si la déduction est de longueur 1, alors ou bien φ appartient à Γ, ou bien c'est un axiome. Par définition de l'ensemble $Th(\mathcal{S})_{\Gamma}$, φ est donc un théorème.

– Cas général : Si la déduction est de longueur n, alors il existe une instance $\frac{\varphi_1,\ldots,\varphi_k}{\varphi}$ d'une règle de $\mathcal{R}$ telle que, pour tout $i \in \{1,\ldots,k\}$, φ_i possède une déduction de longueur strictement inférieure à n. Par hypothèse de récurrence, chaque φ_i est un théorème. D'après la phase d'induction de $Th(\mathcal{S})_{\Gamma}$, φ est donc un théorème également.

$\Leftarrow$ Ce sens se prouve par induction sur l'ensemble des théorèmes.

– Cas de base : Si φ est un théorème de la base, alors une déduction pour φ est (φ).

– Cas général : Sinon, il existe une instance $\frac{\varphi_1,\ldots,\varphi_k}{\varphi}$ d'une règle de $\mathcal{R}$ telle que chaque φ_i est un théorème. Par hypothèse d'induction, chaque φ_i possède donc une déduction $D_i = (\psi_1^i, \ldots, \psi_{k_i}^i, \varphi_i)$ telle que pour tout i, $k_i \geq 0$. La séquence $(D_1, \ldots, D_k, \varphi)$ est donc une déduction de φ.

★

Un théorème de $Th_{\mathcal{S}}(\Gamma)$ s'obtient donc, à partir de Γ, par applications successives de règles d'inférence, c'est-à-dire par *empilement* de ces règles. De ce point de vue, l'obtention d'un théorème de $Th_{\mathcal{S}}(\Gamma)$ peut être mise sous la forme d'un arbre appelé *arbre de preuve*.

Définition 1.2.6 (Arbre de preuve) *Soient $\mathcal{S} = (\mathcal{A}, \mathcal{F}, \mathcal{R})$ un système formel et Γ un ensemble de formules de $\mathcal{F}$. Un* **arbre de preuve** *d'un théorème de $Th_{\mathcal{S}}(\Gamma)$ est un arbre :*

– *dont les feuilles sont les axiomes et les formules de Γ,*

- *dont les nœuds internes sont les théorèmes intermédiaires, tels que le n-uplet constitué des nœuds-fils et de leur nœud-père est une instance d'une règle de $\mathcal{R}$, et*
- *dont la racine est le théorème construit. On appelle cette formule* **conclusion** *de l'arbre, et on note $\pi : \varphi$ lorsqu'un arbre π a pour conclusion φ.*

Exemple 1.2.2 *Nous présentons ici les arbres de preuve de deux tautologies du système formel $\mathcal{S}_{\alpha\beta}$ défini dans la section 1.2.2.3 page 24 :*

$$\dfrac{\dfrac{}{a < e}\, ax2_{ae}}{la < le}\, gd_l \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{d < u}\, ax2_{du}}{do < u}\, g_o}{dow < u}\, g_w}{dow < up}\, d_p}{down < up}\, g_n$$

1.2.3 Le calcul des séquents

Avant de présenter ce système formel, nous devons tout d'abord introduire les notions de séquents et de substitutions.

1.2.3.1 Les séquents

Le calcul que nous étudions ici manipule des phrases que l'on appelle communément des séquents qui ne sont pas des formules mais des expressions construites à partir de celles-ci.

Définition 1.2.7 *Soit Σ une signature. Un* **séquent** *est un couple (Γ, Δ) où Γ et Δ sont des ensembles de Σ-formules finis (qui peuvent être des ensembles vides). On le note $\Gamma \rightarrowtail \Delta$.* [6]

Par convention, nous noterons simplement Γ, φ l'union ensembliste à la place de $\Gamma \cup \{\varphi\}$. Les ensemble de formules seront représentés à l'aide des lettres grecques majuscules $\Gamma, \Delta, \ldots$ et les formules à l'aide des lettres grecques $\varphi, \psi, \ldots$

Intuitivement, un séquent est un moyen de séparer un ensemble d'hypothèses Γ d'un ensemble Δ de conclusions. Sémantiquement, $\Gamma \rightarrowtail \Delta$ signifie que la conjonction $(\varphi_1 \wedge \cdots \wedge \varphi_n)$ des formules de Γ entraîne la disjonction $(\psi_1 \vee \cdots \vee \psi_m)$ des formules de Δ, c'est-à-dire qu'une des formules de Δ doit être vraie dès que

[6] La notation habituellement utilisée pour noter les séquents est $\Gamma \vdash \Delta$, mais nous préférons ici la notation $\Gamma \rightarrowtail \Delta$ pour éviter toute confusion avec le symbole $\vdash$ utilisé pour la notion d'existence d'une déduction dans un système de preuve.

toutes les formules de Γ sont vraies. Ainsi le séquent $\Gamma, \varphi \rightarrowtail \Delta, \varphi$ est toujours vrai.

Remarque 1.2.1 *Le séquent $\varphi_1, \ldots, \varphi_n \rightarrowtail \psi_1, \ldots, \psi_m$, est interprété comme la formule $\varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi_1 \vee \cdots \vee \psi_m$. Le séquent est vrai dans une interprétation ι donnée si la formule associée est vraie dans cette même interprétation. On note*

$$\models^{\iota} \varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi_1 \vee \cdots \vee \psi_m$$

Le séquent est valide si la formule associée est valide, c'est-à-dire vraie pour toute interprétation. On note alors :

$$\models \varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi_1 \vee \cdots \vee \psi_m$$

1.2.3.2 Substitutions

Afin de pouvoir définir les règles de déduction du calcul des séquents, il nous faut définir la notion très importante de substitution. En effet, nous aurons souvent besoin de construire des expressions obtenues à partir d'un terme en remplaçant une variable x par un terme t.

Définition 1.2.8 (Substitution) *Soit Σ une signature. Une* substitution *est une famille indexée par S d'applications $\sigma_s : V_s \rightarrow T_\Sigma(V)_s$.*

Ainsi, une substitution est une interprétation des variables dans les termes. C'est l'équivalent symbolique d'interprétation des variables dans un modèle. Toute substitution $\sigma : V \rightarrow T_\Sigma(V)$ s'étend de façon naturelle à l'ensemble $T_\Sigma(V)$. La fonction obtenue, notée $\sigma^\sharp : T_\Sigma(V) \rightarrow T_\Sigma(V)$, est définie inductivement par :

- $\sigma^\sharp(x) = \sigma(x)$
- $\sigma^\sharp(f(t_1, \ldots, t_n)) = f(\sigma^\sharp(t_1), \ldots, \sigma^\sharp(t_n))$

Cette extension consiste donc à remplacer les variables par leur substitution et conserver tel quel tout le reste.

Dans la suite, nous utiliserons la notation σ à la place de $\sigma^\sharp$. Par convention, les substitutions seront généralement notées à l'aide des lettres grecques minuscules $\sigma, \tau, \ldots$

Lorsque l'ensemble des variables est fini, nous représenterons souvent une substitution σ à l'aide d'un ensemble fini de couples formés d'une variable x_i et d'un terme t_i de même sorte. Cet ensemble sera noté $\{x_1/t_1, \ldots, x_n/t_n\}$, tel que pour tout $i, j, 1 \leq i, j \leq n$, x_i et x_j sont des variables distinctes.

Exemple 1.2.3 *Soit Σ_{Arith} notre exemple de signature, et $V = \{x, y\}$ un ensemble de variable.*
Soient $succ(x)+succ(succ(0))$ un terme de type $entier$, et $\{x/(0+succ(0)), y/succ(0)\}$ une substitution.

$$\sigma(succ(x) + succ(succ(0))) = succ(0 + succ(0)) + succ(succ(0))$$

Dans les formules avec quantificateurs, les variables selon qu'elles sont ou non sous le portée d'un quantificateur sont dites *liées*, ou *libres*.

Définition 1.2.9 *Soit Φ une formule. L'ensemble des* **variables libres** *et l'ensemble des* **variables liées** *de Φ, notés, respectivement, $FV(\Phi)$ et $BV(\Phi)$ sont définis récursivement sur les formules.*

- *Si Φ est une formule atomique, $FV(\Phi)$ est l'ensemble de des variables qui apparaissent au moins une fois dans Φ, et $BV(\Phi) = \emptyset$.*
- *Si Φ est de la forme $(\neg\Psi)$, alors $FV(\Phi) = FV(\Psi)$ et $BV(\Phi) = BV(\Psi)$.*
- *Si Φ est de la forme $(\Psi_1 * \Psi_2)$, avec $* \in \{\wedge, \vee, \Rightarrow\}$ alors $FV(\Phi) = FV(\Psi_1) \cup FV(\Psi_2)$ et $BV(\Phi) = BV(\Psi_1) \cup BV(\Psi_2)$.*
- *Si Φ est de la forme $(Qx.\Psi)$, avec $Q \in \{\forall, \exists\}$, alors $FV(\Phi) = FV(\Psi) - \{x\}$ et $BV(\Phi) = BV(\Psi) \cup \{x\}$.*

Cette notion s'étend naturellement aux ensembles de formules : si $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$ alors $FV(\Gamma) = FV(\varphi_1) \cup \cdots \cup FV(\varphi_n)$ et $BV(\Gamma) = BV(\varphi_1) \cup \cdots \cup BV(\varphi_n)$.

La notion du substitution peut facilement être étendue aux formules, cependant les formules avec quantificateurs posent un problème. En effet, le sens de celles-ci peut être modifié si l'on ne substitue pas correctement les variables. La notion suivante permet de dire si les substitutions sont correctes.

Définition 1.2.10 *Un terme t est* **libre pour une variable** *x dans une formule Φ si :*

- *Φ est une formule atomique,*
- *Φ est de la forme $\Psi_1 * \Psi_2$ avec $* \in \{\wedge, \vee, \Rightarrow\}$, et t est libre pour x dans Ψ_1 et dans Ψ_2,*
- *Φ est de la forme $\neg\Psi_1$, et t est libre pour x dans Ψ,*
- *Φ est de la forme $Qy : s.\Psi$ avec $Q \in \{\forall, \exists\}$ et*
  - *x et y sont la même variable, ou bien*
  - *x et y sont deux variables distinctes et y n'est pas une variable de t,*

  et t est libre pour x dans Ψ.

Dans la suite, lorsque l'on appliquera une substitution $\{x/t\}$ sur une formule Φ, nous supposerons qu'elle remplace toutes les occurrences libres de la variable x dans Φ par un terme t qui est libre pour x dans Φ.

1.2.3.3 Les règles du calcul des séquents

Nous pouvons donc maintenant donner l'ensemble des *règles d'inférence* de notre calcul.

Définition 1.2.11 (Calcul) *Soient Σ une signature, $\Gamma, \Gamma', \Delta, \Delta'$ des ensembles de formules, et Φ, Φ', Ψ des formules. Le* **calcul des séquents** *est composé de l'ensemble des règles d'inférences suivantes :*

$$\frac{}{\Gamma, \Phi \rightarrowtail \Delta, \Phi}\ Ax$$

$$\frac{\Gamma, \Phi, \Phi' \rightarrowtail \Delta}{\Gamma, \Phi \wedge \Phi' \rightarrowtail \Delta}\ \wedge Gauche \qquad \frac{\Gamma \rightarrowtail \Delta, \Phi \qquad \Gamma \rightarrowtail \Delta, \Phi'}{\Gamma \rightarrowtail \Delta, \Phi \wedge \Phi'}\ \wedge Droite$$

$$\frac{\Gamma, \Phi \rightarrowtail \Delta \qquad \Gamma, \Phi' \rightarrowtail \Delta}{\Gamma, \Phi \vee \Phi' \rightarrowtail \Delta}\ \vee Gauche \qquad \frac{\Gamma \rightarrowtail \Delta, \Phi, \Phi'}{\Gamma \rightarrowtail \Delta, \Phi \vee \Phi'}\ \vee Droite$$

$$\frac{\Gamma \rightarrowtail \Delta, \Phi}{\Gamma, \neg\Phi \rightarrowtail \Delta}\ \neg Gauche \qquad \frac{\Gamma, \Phi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta, \neg\Phi}\ \neg Droite$$

$$\frac{\Gamma \rightarrowtail \Delta, \Phi \qquad \Gamma, \Psi \rightarrowtail \Delta}{\Gamma, \Phi \Rightarrow \Psi \rightarrowtail \Delta}\ \Rightarrow Gauche \qquad \frac{\Gamma, \Phi \rightarrowtail \Delta, \Psi}{\Gamma \rightarrowtail \Delta, \Phi \Rightarrow \Psi}\ \Rightarrow Droite$$

Dans les règles avec quantificateurs ci-dessous, on suppose que :
– *x est une variable quelconque,*
– *y est une variable et t un terme, tous deux libres pour x dans Φ*
Dans les règles $\exists Gauche$ et $\forall Droite$ la variable y n'est pas libre dans la conclusion de la règle Γ, Δ ; la variable y est communément appelée eigenvariable[7].

$$\frac{\Gamma, \Phi[t/x], \forall x : s.\Phi \rightarrowtail \Delta}{\Gamma, \forall x : s.\Phi \rightarrowtail \Delta}\ \forall Gauche$$

$$\frac{\Gamma \rightarrowtail \Phi[y/x], \Delta}{\Gamma \rightarrowtail \Delta, \forall x : s.\Phi}\ \forall Droite \quad (y \notin FV(\Gamma, \Delta))$$

[7] La condition de la *eigenvariable* (en allemand) est nécessaire pour la correction du système de preuve. En son absence, des séquents non valides peuvent être prouvés.

$$\frac{\Gamma, \Phi[y/x] \rightarrowtail \Delta}{\Gamma, \exists x : s.\Phi \rightarrowtail \Delta} \exists Gauche \quad (y \notin FV(\Gamma, \Delta))$$

$$\frac{\Gamma \rightarrowtail \Phi[t/x], \Delta}{\Gamma \rightarrowtail \Delta, \exists x : s.\Phi} \exists Droite$$

$$\frac{\Gamma \rightarrowtail \Delta, \Phi \qquad \Gamma', \Phi \rightarrowtail \Delta'}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'} Coupure$$

Le nom de chaque règle est donné à droite de la barre horizontale de chaque règle. Au dessus de cette barre se trouvent souvent un ou deux séquents qui sont les *prémisses* de la règle et en dessous un séquent appelé *conclusion*. La seule règle sans prémisse représente les axiomes, c'est-à-dire les séquents qui sont toujours valides.

Comme nous le verrons dans le calcul du chapitre 3 page 83, nous pouvons très bien nous passer des règles $\wedge Gauche$, $\wedge Droite$, $\Rightarrow Gauche$, $\Rightarrow Droite$, $\forall Gauche$, et $\forall Droite$. Par définition, les connecteurs $\Rightarrow$ et $\wedge$ peuvent s'écrire à l'aide des deux connecteurs $\vee$ et $\neg$, et le quantificateur $\forall$ peut être défini à l'aide de $\exists$ et $\neg$.

Ce calcul peut être immédiatement représenté par le système formel $S_{LK} = (A_{LK}, F_{LK}, R_{LK})$ où, étant donnée une signature du premier ordre $\Sigma = (S, F, R, V)$, on a :

- A_{LK} est l'alphabet $F \cup R \cup V \cup \{\neg, \wedge, \vee, \forall, \exists\} \cup \{\rightarrowtail\}$,
- F_{LK} contient tous les séquents bien formés que l'on peut construire à partir de A_{LK},
- R_{LK} est l'ensemble des instances des schémas de règles donné dans le calcul précedent.

Dans la présentation classique du calcul des séquents **LK**, il existe plusieurs règles d'inférences dites structurelles. Celles-ci sont implicites dans le calcul que nous venons de présenter parce que les séquents sont construits à partir d'ensemble de formules et non de séquences de formules comme cela est fait habituellement. L'utilisation des ensembles peut être vue comme une optimisation de la version du calcul des séquents avec des séquences. À titre d'information, voici ces règles structurelles. Étant donné un séquent $\Gamma \rightarrowtail \Delta$, pour ce calcul, Γ et Δ sont maintenant des séquences finies de formules (nous aurions pu également choisir de considérer des multi-ensembles).

Contraction

$$\frac{\Gamma, \Phi, \Phi \rightarrowtail \Delta}{\Gamma, \Phi \rightarrowtail \Delta}\ Cont\ Gauche \qquad \frac{\Gamma \rightarrowtail \Phi, \Phi, \Delta}{\Gamma \rightarrowtail \Phi, \Delta}\ Cont\ Droite$$

Affaiblissement

$$\frac{\Gamma \rightarrowtail \Delta}{\Gamma, \Phi \rightarrowtail \Delta}\ Aff\ Gauche \qquad \frac{\Gamma \rightarrowtail \Delta}{\Gamma \rightarrowtail \Phi, \Delta}\ Aff\ Droite$$

Échange

$$\frac{\Gamma, \Phi, \Phi' \rightarrowtail \Delta}{\Gamma, \Phi', \Phi \rightarrowtail \Delta}\ Ech\ Gauche \qquad \frac{\Gamma \rightarrowtail \Phi, \Phi', \Delta}{\Gamma \rightarrowtail \Phi', \Phi, \Delta}\ Ech\ Droite$$

1.3 La logique équationnelle

Nous allons maintenant nous intéresser à une restriction de la logique des prédicats du premier ordre bien adaptée à la spécification de logiciels. Cette restriction est appellée logique équationnelle. Les formules atomiques sont de simples équations entre termes. Elle est très utilisée lorsque les logiciels sont vus au travers des types de données qu'ils manipulent.

1.3.1 Présentation

Dans sa présentation, la logique équationnelle diffère très peu de la logique des prédicats du premier ordre. Syntaxiquement, les restrictions portent sur les signatures où l'on retire l'ensemble R des symboles de prédicats. Les formules sont alors construites comme précédemment excepté pour les formules atomiques qui sont des équations. Ce sont des couples (t, t') construits à partir d'une signature restreinte $\Sigma = (S, F, V)$ où t et t' sont des termes avec variables de même type. On a l'habitude de noter de tels couples $t = t'$.

Sémantiquement, un Σ-modèle associé à une signature équationnelle est défini comme dans la définition 1.1.6 sans l'interprétation des prédicats autre que le prédicat binaire de l'égalité. On parle alors de Σ-algèbres plutôt que de modèles du premier ordre. L'évaluation des termes et la satisfaction des formules suivent en tout point les sections 1.1.2.2 et 1.1.2.3.

Afin de prendre en compte le type de raisonnement concis et efficace attaché à l'égalité et connu sous le nom de remplacement d'égal par égal, nous étendons le calcul des séquents, précédemment défini, au raisonnement équationnel. Voici une spécialisation du calcul des séquents (voir [Gal86]) auquel nous ajoutons les

règles suivantes telles que $t_1, \ldots, t_n, t'_1, \ldots, t'_n, t$ sont des termes, et $s_1, \ldots, s_n, s$ des noms de sortes :

$$\frac{\Gamma, t =_s t \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta}\ Refl_{Eq}$$

$$\frac{\Gamma, (t_1 =_{s_1} t'_1 \wedge \cdots \wedge t_n =_{s_n} t'_n \Rightarrow f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)) \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta}\ Rempl_{Eq}$$

$$\frac{\Gamma, (t_1 =_s t_2 \wedge t_2 =_s t_3 \Rightarrow t_1 =_s t_3) \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta}\ Trans_{Eq}$$

$$\frac{\Gamma, (t_1 =_s t_2 \Rightarrow t_2 =_s t_1) \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta}\ Sym_{Eq}$$

1.3.2 Formules équationnelles simplifiées

1.3.2.1 Les formules conditionnelles positives

L'une des restrictions de la logique équationnelle a été particulièrement étudiée pour ses bonnes propriétés algébriques et de décidabilité (principalement pour des techniques de réécriture - voir le chapitre suivant). La restriction porte sur les formules considérées. Dans le cadre de cette logique, les formules sont de la forme :

$$t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow t = t'$$

Elles permettent de disposer de pré-conditions lors de la définition d'une opération, et elles sont construites à l'aide des équations et des deux seuls connecteurs logique $\wedge$ et $\Rightarrow$.

On parle alors de *formules conditionnelles positives*. Ce sont les clauses de Horn de la logique équationnelles. Ces formules seront très utiles dans le chapitre 6 où nous considérerons une classe de spécifications algébriques pour laquelle les axiomes sont des formules conditionnelles positives.

Pour ce type de formules, on peut définir un calcul mieux adapté. Ce dernier est défini par un ensemble des règles d'inférences. Celui-ci se divise en deux groupes : les règles définissant le prédicat égalité de façon générale et les règles logiques définissant les connecteurs $\wedge$ et $\Rightarrow$ et le quantificateur universel implicite $\forall$.

Définition 1.3.1 *Soient $\Sigma = (S, F, V)$ un signature et Γ un ensemble de* **formules conditionnelles positives.** *Soient $\alpha, \beta, \alpha_1, \ldots, \alpha_m$ des équations. Soient $t, t', t'', t_1, \ldots, t_n, u, v, t'_1, \ldots, t'_n$ des termes de $T_\Sigma(V)$.*

Le système de preuve est défini par les règles d'inférences[8] *suivantes :*

$$\textbf{Ref}\frac{}{\Gamma \vdash t = t} \qquad \textbf{Axiom}\frac{}{\Gamma \cup \varphi \vdash \varphi}$$

$$\textbf{Trans}\frac{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow t = t' \qquad \Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow t' = t''}{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow t = t''}$$

$$\textbf{Rempl}\frac{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow t_1 = t'_1 \qquad \ldots \qquad \Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow t_n = t'_n}{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}$$

$$\textbf{Sym}\frac{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow t = t'}{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow t' = t} \qquad \textbf{Subst}\frac{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow \alpha}{\Gamma \vdash \bigwedge_{1\leq i\leq m}\sigma(\alpha_i) \Rightarrow \sigma(\alpha)}$$

$$\textbf{Mon}\frac{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow \alpha}{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \wedge \beta \Rightarrow \alpha}$$

$$\textbf{MP}\frac{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \wedge u = v \Rightarrow \alpha \qquad \Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow u = v}{\Gamma \vdash \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow \alpha}$$

Comme ce système de preuve est également un système formel, une preuve d'une formule φ conditionnelle positive à partir d'un ensemble d'axiomes Ax est un arbre de conclusion $Ax \vdash \varphi$ dont les feuilles sont soit un axiome (**Axiom**) soit une instance de la réflexivité (**Ref**), et dont les noeuds sont construits à l'aide d'instances des autres règles :

[8] Nous ne sommes plus dans le cadre du calcul des séquents, donc nous utilisons désormais la notation classique pour le prédicat $\vdash$

- **Trans**, la transitivité ;
- **Rempl** , le remplacement (ou contexte) ;
- **Sym** , la symétrie ;
- **Subst** , la substitution ;
- **Mon** , la monotonie ;
- **MP** , le modus ponens[9], qui est une règle selon laquelle si une formule α implique une formule β, alors on peut déduire que si α est vraie alors β l'est également.

La règle de tautologie n'est pas écrite explicitement ici, en revanche on peut prouver des formules de la forme $\alpha \Rightarrow \alpha$ si l'on considère que toutes les formules de ce type sont contenues dans l'ensemble Γ des formules (de manière implicite).

1.3.2.2 La logique équationnelle élémentaire

Nous parlerons de logique équationnelle élémentaire lorsque les formules sont réduites à de simples équations. Il s'agit d'une restrictions des formules conditionnelles positives où l'on utilise vraiment aucun connecteur logique. Il est possible dans ce cas de définir un système de preuve dédié où l'on ne manipule que des équations. Nous étudierons en détail le système de preuve pour ces équations dans la section 3.3.1.1 page 74.

1.4 Les spécifications algébriques

1.4.1 Préliminaires

Les *spécifications algébriques* sont une technique permettant de définir le comportement (c'est-à-dire spécifier) des logiciels au travers des structures de données qu'ils manipulent (les listes, les files, les piles, les arbres, etc.).

La manière la plus commune de caractériser une spécification SP est de donner un ensemble d'axiomes Ax composé de formules de la logique des prédicats du premier ordre. On note alors $SP = (\Sigma, Ax)$ et on dit que SP est une *spécification axiomatique*. Formellement, les spécifications se définissent de la façon suivante :

Définition 1.4.1 (Spécification) *Une* **spécification** SP *est une paire* (Σ, Ax) *où* Σ *est une signature du premier ordre et* Ax *un ensemble de* Σ*-formules. Ces formules de* Ax *sont appelées* **axiomes** *de la spécification* SP*.*

[9] Modus Ponens : locution latine signifiant "mode qui pose"

On classe ces spécifications en fonction de la forme des axiomes. Celle-ci dépend de ce que l'on veut définir comme propriétés. Plus la classe d'axiomes considérée est large, plus le pouvoir d'expression de nos spécifications est grand, mais en contrepartie l'exploitation des spécifications peut devenir plus difficile.
Par exemple, pour définir l'addition sur les entiers, on n'a pas besoin d'avoir des axiomes très compliqués. Des équations suffisent :

$$\begin{array}{rcl} 0 + n & = & 0 \\ succ(n) + m & = & succ(n + m) \end{array}$$

Par contre pour définir des opérations plus complexes, on aura besoin d'un langage plus expressif. Par exemple pour définir l'opération modulo sur les entiers naturels, nous nous servons d'une opération $<$ en pré-condition.

$$\begin{array}{rcrcl} n < m = true & \Rightarrow & modulo(n, m) & = & n \\ n < m = false & \Rightarrow & modulo(n, m) & = & modulo(n - m, m) \end{array}$$

On reconnaît ici des formules conditionnelles positives de la forme $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow t = t'$. Lorsque tous les axiomes d'une spécification sont de cette forme, on parle alors de spécification conditionnelle positive.

Il existe des nombreux langages de spécification. Dans cette thèse nous allons considérer une classe de spécifications dites algébriques [Wir90, AKKB99]. Elles ont été utilisés aussi bien dans le cadre de travaux théoriques sur le test [AAB+05a, ALG02, Mac00a, MS02, Mac00b, DW00, ALGM96, LGA96], que pour construire des outils de génération de test [CH00, BW05c, Mar91b].

Dans cette thèse nous nous intéresserons plus spécifiquement aux spécifications conditionnelles positives qui constituent un formalisme simple et couramment utilisé notamment dans le domaine du test [BGM91, Gau95, Mar91b].

Ces spécifications permettent de représenter de nombreuses structures de données différentes. En voici deux exemples :

Exemple 1.4.1 *À l'aide de la signature Σ_{Arith} donnée dans l'exemple 1.1.1 de la présentation de la logique du premier ordre, nous pouvons associer l'ensemble Ax_{Arith} des axiomes suivant :*

$$\begin{array}{rcll} 0 + x & = & x & \textit{(Add1)} \\ succ(x) + y & = & succ(x + y) & \textit{(Add2)} \\ x * 0 & = & 0 & \textit{(Mult1)} \\ x * succ(y) & = & x + succ(x * y) & \textit{(Mult2)} \end{array}$$

La paire $(\Sigma_{Arith}, Ax_{Arith})$ définit la spécification SP_{Arith} de l'arithmétique élémentaire.

Exemple 1.4.2 *Pour spécifier les listes d'éléments (le type élément et un type générique qui peut être instancié par n'importe quel type selon les besoins), nous utiliserons la signature équationnelle suivante* $\Sigma_{Listes} = (S_{Listes}, F_{Listes}, V_{Listes})$ *telle que :*

$$
\begin{array}{rcl}
S_{Listes} & = & \{liste, elem, entier\} \\
F_{Listes} & = & \{[] :\rightarrow liste \\
 & & \_ :: \_ : elem \times liste \rightarrow liste, \textit{ (ajout d'un élément dans une liste)} \\
 & & \_@\_ : liste \times liste \rightarrow liste, \textit{ (concaténation de deux listes)} \\
 & & long : liste \rightarrow entier, \textit{ (longueur d'une liste)} \\
 & & 0 :\rightarrow entier, \\
 & & succ : entier \rightarrow entier\} \textit{ (passage au successeur)} \\
V_{Listes} & = & \{l, l' : liste, e : elem, x, y : entier\}
\end{array}
$$

La spécification des listes SP_{Listes} *se définit à l'aide de la signature* Σ_{Listes} *que nous venons de définir et de l'ensemble d'axiomes* Ax_{Listes} *suivant :*

$$
\begin{array}{rcl}
[]@l & = & l \\
(e :: l)@l' & = & e :: (l@l') \\
long([]) & = & 0 \\
long(e :: l) & = & succ(long(l))
\end{array}
$$

L'intérêt des formalismes de spécification est de permettre d'écrire des spécifications abstraites, c'est-à-dire s'intéressant au *quoi* du logiciel (c'est-à-dire qu'est-ce que le logiciel est supposé faire) mais pas au *comment* (c'est-à-dire comment il est supposé le faire). L'avantage d'une spécification abstraite est qu'elle est souvent plus concise et plus claire que l'écriture à l'aide d'un langage de programmation. C'est lors des étapes successives de raffinement que les choix d'algorithmes devront être faits permettant ainsi d'aborder cet aspect du "comment".

1.4.2 Conséquences sémantiques

Dans les spécifications, nous n'écrivons que les propriétés requises d'un logiciel pour qu'il réponde au problème demandé à l'aide des axiomes. Cependant, de ces propriétés élémnetaires requises, nous pouvons déduire un ensemble (souvent infini) de propriétés non-explicitées dans la spécification. On parle alors de *conséquences sémantiques.*

Définition 1.4.2 *Soit* $SP = (\Sigma, Ax)$ *une spécification. Un* Σ*-modèle* $\mathcal{M}$ *est un* SP**-modèle** *(ou encore un modèle de la spécification* SP*) si et seulement si pour tout axiome* $\varphi \in Ax$, $\mathcal{M} \models \varphi$.

Une Σ-formule ψ est une **conséquence sémantique** *de SP, notée $SP \models \psi$, si et seulement si pour chaque SP-modèle $\mathcal{M}$, nous avons $\mathcal{M} \models \psi$.*

$SP^\bullet$ est l'ensemble de toutes les conséquences sémantiques d'une spécification SP.

On remarque que $SP^\bullet$ contient nécessairement les axiomes de l'ensemble Ax de la spécification.

Exemple 1.4.3 *Nous voulons montrer que l'égalité très simple $x + succ(0) = succ(x)$ est une conséquence sémantique de la spécification SP_{Arith} donnée dans l'exemple 1.4.1 page 15. Cela revient à montrer que $SP_{Arith} \models x + succ(0) = succ(x)$. Soit $\mathcal{M}$ un SP_{Arith}-modèle quelconque. Soit ι une interprétation des variables qui à x associe un entier n quelconque. Montrons alors que $\mathcal{M} \models_\iota x + succ(0) = succ(x)$. Les choix de $\mathcal{M}$ et ι étant quelconques cela reviendra à démontrer $SP_{Arith} \models x + succ(0) = succ(x)$. Par hypothèse, les égalités suivantes sont vérifiées :*

$n +^{\mathcal{M}} succ^{\mathcal{M}}(0^{\mathcal{M}}) = succ^{\mathcal{M}}(n +^{\mathcal{M}} 0^{\mathcal{M}})$ *(à l'aide de l'axiome Add2)*
$succ^{\mathcal{M}}(n +^{\mathcal{M}} 0^{\mathcal{M}}) = succ^{\mathcal{M}}(n)$ *(à l'aide de l'axiome Add1)*

Nous avons bien l'égalité de la formule de départ. C'est bien une conséquence sémantique de la spécification.

La notion de conséquence sémantique permet ainsi de considérer une large classe de formules qui sont des conséquences directes d'une spécification.

1.4.3 Autre exemple : les piles

De nombreuses structures de données peuvent être décrites à l'aide des spécifications. Par exemple, les piles :

Soit $\Sigma_{Piles} = (S_{Piles}, F_{Piles}, V_{Piles})$ une signature équationnelle telle que :

$$\begin{aligned}
S_{Piles} &= \{entier, pile\} \\
F_{Piles} &= \{0 : entier, succ : entier \to entier, \\
&\quad vide :\to pile, \\
&\quad empiler : entier \times pile \to pile, \\
&\quad depiler : pile \to pile, \\
&\quad haut : pile \to entier, \\
&\quad hauteur : pile \to entier\} \\
V_{Piles} &= \{x : entier, p : pile\}
\end{aligned}$$

Les opérations $vide$ et $empiler$ sont les constructeurs de pile. L'opération $depiler$ permet d'enlever le dernier élément empilé dans une pile. L'opération $haut$ renvoie ce dernier élément sans le supprimer. L'opération $hauteur$ renvoie la hauteur de la pile (le nombre d'éléments).

La spécifications des piles $Piles$ est composé de la signature Σ_{Piles} et de l'ensemble Ax_{Piles} d'axiomes suivant :

$$\begin{aligned} depiler(vide) &= vide \\ depiler(empiler(x,p)) &= p \\ haut(vide) &= 0 \\ haut(empiler(x,p)) &= x \\ hauteur(vide) &= 0 \\ hauteur(empiler(x,p)) &= succ(hauteur(p)) \end{aligned}$$

Chapitre 2

Réécriture

Ce deuxième chapitre est dédié à une technique plus opérationnelle de raisonnement que les systèmes d'inférence : la réécriture. La réécriture est une branche de l'informatique théorique qui combine des éléments de logique, de programmation fonctionnelle, d'algèbre, et de preuve automatique. On lui trouve de nombreuses applications en génie logiciel, notamment en test fonctionnel, et plus généralement dès que l'on est en présence d'équations. Nous parlerons ici essentiellement de la réécriture de termes [BN98, Klo92, DJ90] qui définit une sémantique opérationnelle de la logique équationnelle (voir section 1.3).

Succinctement, on distingue la réécriture du raisonnement équationnel présenté dans le chapitre précédent par le fait que les éléments manipulés dans les preuves ne sont plus les équations mais directement les termes. Ceci est rendu possible par la manipulation de règles de réécriture qui sont des couples de termes orientés de la gauche vers la droite. À l'aide de ces règles, la partie gauche d'une règle peut être remplacée par la partie droite dans tout terme où cette partie gauche apparaît comme sous-terme. Pour ce qui nous concerne, l'intérêt de cette technique vient principalement du fait que nous utiliserons des outils formels principalement issus de cette technique. Ceux-ci nous permettront de définir et démontrer la complétude des techniques de sélection de jeux de tests développées dans le chapitre 6 de cette thèse. Plus précisément, nous allons utiliser dans la suite de ce manuscrit deux outils formels issus de la réécriture : l'unification, et les ordres de simplification. Mais avant de présenter ces deux techniques, nous devons au préalable présenter quelques notions utiles à ces deux techniques.

2.1 Positions et contexte

Les termes de la logique des prédicats du premier ordre, de par leur définition inductive, peuvent être représentés à l'aide d'un arbre où les noeuds sont les

symboles d'opérations, et les feuilles les variables ou les constantes. Il est ainsi possible de repérer chaque élément grâce à sa position dans le terme (dans l'arbre) définie comme un mot sur $\mathbb{N}$.

Définition 2.1.1 (Position) *Soient Σ une signature, V un ensemble de variables, et t un terme de $T_\Sigma(V)$.*

L'ensemble des **positions** *d'un terme t est un ensemble $Pos(t)$ de chaînes de caractères sur l'alphabet des entiers positifs qui est défini de la manière suivante :*

- *si $t \in V$ alors $Pos(t) = \{\varepsilon\}$ où ε dénote la chaîne de caractères vide,*
- *si $t = f(t_1, \ldots, t_n)$, alors*

$$Pos(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{i.p \mid p \in Pos(t_i)\}$$

.

Pour toute position $\omega \in Pos(t)$, on note $-|_\omega : T_\Sigma(V) \to T_\Sigma(V)$ l'application partielle définie inductivement pour tout terme $t \in T_\Sigma(V)$ sur la taille des mots de $Pos(t)$ par :

- $t|_\varepsilon = t$
- $t|_{n.\omega'} = \begin{cases} t_n|_{\omega'} \ si \ t = f(t_1, \ldots, t_k) \ avec \ k \geq n, \\ ind\acute{e}fini \ sinon \end{cases}$

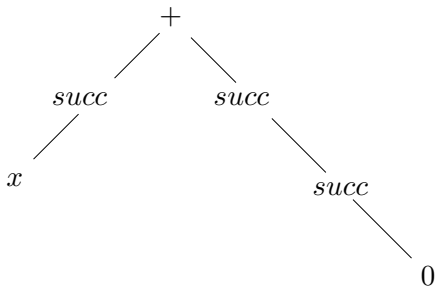
La position ε est la position de la **racine** *du terme t. Pour toute position ω définie pour un terme t, $t|_\omega$ est appelé* **sous-terme** *de t.*

Définition 2.1.2 (Contexte) *Soient Σ une signature, V un ensemble de variables, et $\Box$ une constante n'appartenant pas à Σ. Un* **contexte** *pour Σ ou* **Σ-contexte** *est un terme de $T_{\Sigma \cup \{\Box^0\}}(V)$ dans lequel il n'y a qu'une seule occurrence du symbole de constante $\Box$.*

Dans la suite, on notera $s[.]_\omega$ pour désigner un contexte s où l'unique occurrence de $\Box$ apparaît à la position ω.

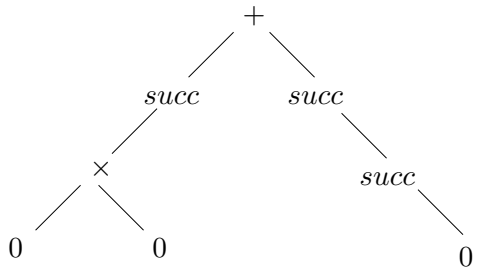
Enfin, pour tout terme t et tout contexte $s[.]_\omega$, on notera $s[t]_\omega$ le terme de $T_\Sigma(V)$ obtenu à partir de s en remplaçant le symbole $\Box$ par le terme t à la position ω (c'est-à-dire tel que $s[t]_\omega|_\omega = t$).

Exemple 2.1.1 *Soit $t = succ(x) + succ(succ(0))$ un terme. L'arbre associé est de la forme suivante :*



Soit s le contexte $succ(\Box) + succ(succ(0))$ obtenu à partir du terme t en remplaçant x par le symbole $\Box$. Nous pouvons ainsi construire un nouveau terme en remplissant $\Box$ qui représente une position "vide" dans le terme. Nous pouvons le remplacer, par exemple, par un terme 0×0.

On notera le terme obtenu $s[0 \times 0]_{1.1} = succ(0 \times 0) + succ(succ(0))$, ce qui nous donne l'arbre suivant :



2.2 Unification

2.2.1 Présentation

Le concept d'unification [Lal90] est une notion centrale en programmation logique. Il permet aussi bien de prouver que de calculer. Il permet de résoudre et de propager des contraintes. Il nous sera très utile dans la suite de cette thèse, notamment pour définir nos procédures de dépliage. Dans toute cette section, nous allons unifier des termes de la logique des prédicats du premier ordre.

Résoudre une équation $t_1 = t_2$, c'est chercher ses solutions. Cela revient à trouver dans une algèbre $\mathcal{M}$ des valeurs pour les variables de t_1 et t_2, qui satisfont cette équation.

Définition 2.2.1 *Soient Σ une signature, t_1, t_2 des termes dans $T_\Sigma(V)$, et $\mathcal{M}$ une Σ-algèbre. Toute évaluation $\iota : Var(t_1) \cup Var(t_2) \to M$ telle que $\iota(t_1) = \iota(t_2)$, est une* **solution de l'équation** *$t_1 = t_2$ dans $\mathcal{M}$.*

Exemple 2.2.1 *L'équation $x^2 - 3 = 0$ n'a pas de solution dans $\mathbb{N}$, mais a deux solution dans $\mathbb{R}$ ($\iota(x) = \sqrt{3}$ et $\iota(x) = -\sqrt{3}$).*

L'énoncé du problème fait intervenir l'algèbre $\mathcal{M}$ dont dépend l'existence et le nombre des éventuelles solutions.

Maintenant, on peut chercher des solutions dans l'algèbre des termes avec variables $T_\Sigma(V)$. Dans ce cas là, on parle d'**unification finie**. Dans sa forme générale, l'unification se définie pour un ensemble d'équations.

Définition 2.2.2 (Unificateur) *Soient $t_1, \ldots t_n, s_1, \ldots, s_n$ des termes de $T_\Sigma(V)$. Un* **problème d'unification** *est un ensemble P d'équations de la forme $\{t_1 = s_1, \ldots, t_n = s_n\}$. Une substitution σ est un* **unificateur** *de P si pour tout $i, 1 \leq i \leq n$, on a $\sigma(t_i) = \sigma(s_i)$. On note $U(P)$ l'ensemble des unificateurs de P.*

P est **unifiable** *si $U(P) \neq \emptyset$.*

Deux termes r et s sont unifiables par la substitution σ si σ est un unificateur pour le problème d'unification $\{r = s\}$. De même, deux formules atomique $p(t_1, \ldots, t_n)$ et $p(s_1, \ldots, s_n)$, où p est un symbole de prédicat, sont unifiables par σ si σ est un unificateur pour le problème d'unification $\{t_1 = s_1, \ldots, t_n = s_n\}$.

2.2.2 Unificateur principal

Étant donné un problème d'unification P, plusieurs unificateurs de P différents peuvent exister. Considérons le problème d'unification $\{f(x) = f(g(z))\}$. $\sigma_1 = [x/g(a), z/a]$ est un unificateur car on a bien $\sigma_1(f(x)) = \sigma_1(f(g(z))) = f(g(a))$, mais ce n'est pas le seul. En effet, la substitution $\sigma_2 = [x/g(z)]$ est également un unificateur.

Afin de comparer ces substitutions, on considère les définitions suivantes :

Définition 2.2.3 (Unificateur principal) *Soient σ et θ deux substitutions. θ est dite* **plus générale** *que σ, et on note $\sigma \geq \theta$ s'il existe une substitution μ telle que $\sigma = \mu \circ \theta$.*

Soit P un problème d'unification et soit σ un unificateur de P. σ est un **unificateur principal** *(most general unifier ou* **mgu** *en anglais) de P, si pour tout unificateur σ' de P, il existe une substitution γ telle que $\sigma' = \sigma \circ \gamma$.*

L'unificateur principal est une substitution plus générale que toutes les autres. Il est plus général car il impose le minimum de contraintes possibles sur les substitutions des variables. Dans notre exemple ci-dessus , l'unificateur $\sigma_2 = [x/g(z)]$ est non seulement un unificateur de notre problème, mais il est plus général car $\sigma_1 > \sigma_2$ et il n'existe pas d'autres substitution σ_n telle que $\sigma_2 > \sigma_n$.

Sous les conditions ci-dessous, on peut montrer que si P a un unificateur, alors il a un **unificateur principal** unique. L'unificateur principal est, dans ce cas là, noté $mgu(P)$. Voici les deux conditions que l'on impose :

1. On identifie deux unificateurs σ et σ' d'un problème P qui ne diffèrent que par des renommages de variables.
2. Si P est un problème d'unification et σ un unificateur principal de P, alors, si le couple x/t est un élément de σ, x apparaît forcément dans une équation de P.

Ces deux conditions sont élémentaires et l'on peut dire que le résultat ci-dessus est vrai pour tout problème d'unification.

2.2.3 Algorithmes d'unification

Une autre propriété fondamentale de l'unificateur principal est que s'il existe, il est calculable pour tout problème d'unification.

Il existe plusieurs algorithmes d'unification permettant directement de décider si deux termes sont unifiables ou pas (c'est-à-dire de vérifier que $U(P) \neq \emptyset$, et, dans le cas où l'unificateur de P existe, de calculer $mgu(P)$).

Ici, nous présentons le premier algorithme d'unification proposée par Robinson [Rob65] Il permet directement de décider si deux formules atomiques $p(t_1, \ldots, t_n)$ et $p(s_1, \ldots, s_n)$ sont unifiables ou pas (p étant un symbole de prédicat). Il permet donc de résoudre "naïvement" tout problème d'unification de la forme $\{t_1 = s_1, \ldots, t_n = s_n\}$. Les formules atomiques y sont vues comme deux mots que l'on lit et traite de gauche vers la droite. On note ε la substitution vide.

Algorithme d'unification de Robinson Soit A_1 et A_2 deux formules atomiques.

1. $\sigma = \varepsilon$
2. **Tant que** $\sigma(A_1) \neq \sigma(A_2)$ faire
   (a) Soit ω la position du symbole d'opération le plus à gauche de $\sigma(A_1)$ différent du symbole de même position dans A_2,
   (b) soit $t_1 = \sigma(A_1)|_\omega$ et $t_2 = \sigma(A_2)|_\omega$ les sous-termes qui commencent à ce symbole,
   (c) **si** ni t_1, ni t_2 ne sont des variables ou bien un parmi t_1, t_2 est une variable qui est un sous-terme strict[1] de l'autre **alors** A_1 et A_2 ne sont

[1] Un sous-terme strict d'un terme t est un terme qui apparaît dans t mais qui n'est pas t.

pas unifiables ;
sinon déterminer x, une variable qui est ou bien t_1 ou bien t_2, et t qui est l'autre parmi t_1, t_2.
$\sigma = \sigma \cup \{x/t\}$.
findusi

findutantque

3. σ est l'unificateur le plus général de A_1 et A_2.

Autre algorithme d'unification Il existe un autre algorithme d'unification, plus élégant, et qui utilise des règles de réécriture [Lal90].

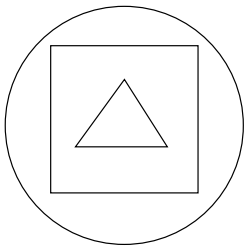
2.3 Systèmes de réécriture

2.3.1 Réécriture de termes

La réécriture de termes est une technique de raisonnement qui consiste à réduire une expression en la remplaçant par une autre (par exemple pour la simplifier).

Plaçons nous dans le cadre des termes de la logique du premier ordre (c'est-à-dire $T_\Sigma(V)$). Considérons un type $figure$ qui est construit de la manière suivante :
- $rien$ est une constante de type $figure$,
- $carre$, $triangle$ et $cercle$ trois opérations qui prennent en argument un terme de type $figure$ et qui renvoient un type $figure$.

Le terme $cercle(carre(triangle(rien)))$ est de type figure, on le représente par une imbrication de figures :

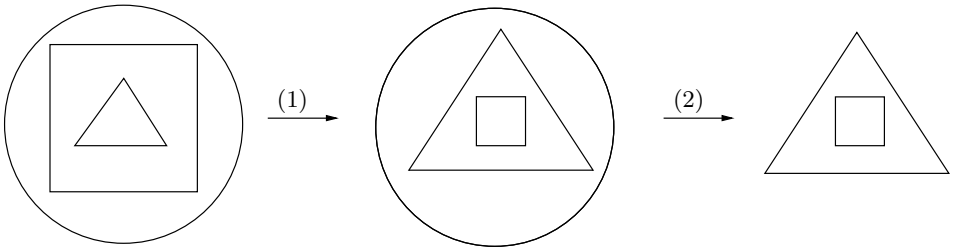


Une *règle de réécriture* pour ces figures est une expression qui transforme une figure en une autre figure. Par exemple, on pourrait avoir l'ensemble composé des deux règles suivantes :

$carre(triangle(f)) \xrightarrow{(1)} triangle(carre(f))$

$cercle(triangle(f)) \overset{(2)}{\rightarrow} triangle(f)$
avec f une variable.

Cet ensemble, appelé *système de réécriture de termes*, nous permet de transformer notre figure de départ de la manière suivante :



Comme nous le constatons, ces règles peuvent s'appliquer à l'intérieure d'une figure, et la variable f des règles peut être remplacée par n'importe quelle figure. On ne pouvait pas appliquer la règle (2) avant la règle (1) dans cet exemple. On ne peut plus appliquer de règles sur la figure $triangle(carre(rien))$.

Un système de réécriture est donc un ensemble de règles de la forme $a \rightarrow b$. Ces règles sont utilisées de manière répétitive pour obtenir la forme la plus simple possible. Chaque étape consistant à remplacer les sous-termes d'une expression donnée apparaissant comme membre gauche d'une règle de réécriture par le membre droit de cette même règle. De manière plus formelle on définit ce système ainsi :

Définition 2.3.1 (Règle et système de réécriture) *Soit $\Sigma = (S, F, V)$ une signature équationnelle. Soient l et r des termes de $T_\Sigma(V)$. Une* **règle de réécriture** *est une paire (l, r) de $T_\Sigma(V) \times T_\Sigma(V)$ telle que $Var(r) \subseteq Var(l)$ On notera cette paire $l \rightarrow r$.*
Un ensemble de règles de réécriture $\mathcal{R}$ est appelé **système de réécriture de termes**.

Exemple 2.3.1 *Comme exemple de système de réécriture, nous pouvons prendre la définition de la fonction récursive d'Ackermann sur les entiers naturels.*

Pour spécifier une fonction telle que celle d'Ackermann nous aurons besoin de la signature suivante $\Sigma_{Ack} = (S_{Ack}, F_{Ack}, V_{Ack})$ telle que :

- $S_{Ack} = \{entier\}$,
- $F_{Ack} = \{0 : entier, succ : entier \rightarrow entier, Ack : entier \times entier \rightarrow entier\}$,
- $V_{Ack} = \{x, y : entier\}$.

La fonction d'Ackermann peut alors se définir par le système $\mathcal{R}_{Ack}$ composé des trois règles de réécriture suivantes :

$$
\begin{array}{lrcl}
\textit{(1)} & Ack(0, y) & \rightarrow & succ(y) \\
\textit{(2)} & Ack(succ(x), 0) & \rightarrow & Ack(x, succ(0)) \\
\textit{(3)} & Ack(succ(x), succ(y)) & \rightarrow & Ack(x, Ack(succ(x), y))
\end{array}
$$

Les systèmes de réécriture définissent un ensemble de règles de base à appliquer sur les termes. L'application d'une seule de ces règles sur un terme est appelée *étape de réécriture* :

Définition 2.3.2 (Étape de réécriture) *Soit $\mathcal{R}$ un système de réécriture. On note $\rightarrow_{\mathcal{R}} \subseteq T_\Sigma(V) \times T_\Sigma(V)$ la plus petite relation (au sens de l'inclusion) définie par : $t \rightarrow_{\mathcal{R}} t'$ si et seulement s'il existe*

- $u \rightarrow v \in \mathcal{R}$,
- $\sigma : V \rightarrow T_\Sigma(V)$,
- et un contexte $s[.]_\omega$

tels que $t = s[\sigma(u)]_\omega$ et $t' = s[\sigma(v)]_\omega$.

On dit que t se **réécrit** *en t', et on note $t \rightarrow_{\mathcal{R}} t'$.*

Exemple 2.3.2 *Pour le système de réécriture $\mathcal{R}_{Ack}$ de la fonction d'Ackermann, le terme $Ack(succ(0), succ(0))$ peut se réécrire de la manière suivante :*

- *en $Ack(0, Ack(succ(0), 0))$ grâce à la règle (3),*
- *en $Ack(0, Ack(0, succ(0)))$ grâce à la règle (2) appliquée sur le sous-terme $Ack(succ(0), 0)$,*
- *en $Ack(0, succ(succ(0)))$ grâce à la règle (1) appliquée sur le sous-terme $Ack(0, succ(0))$,*
- *en $succ(succ(succ(0)))$ (c'est-à-dire 3) grâce à la règle (1).*

2.3.2 Propriétés des systèmes de réécriture

Comme nous l'avons vu dans l'exemple d'introduction, les règles de réécriture peuvent être appliquées de manière répétitive. On parle alors de dérivation :

Définition 2.3.3 (Dérivation) *Avec les notations de la définition 2.3.2, une* **dérivation** *est une séquence de termes de $T_\Sigma(V)$, $(t_1, \ldots, t_n)$ telle que $t_i \rightarrow_{\mathcal{R}} t_{i+1}$ pour tout $i \in \{1, \ldots, n-1\}$.*

L'application successive de plusieurs étapes de réécriture est également notée $\overset{*}{\rightarrow}_{\mathcal{R}}$. Elle correspond à la fermeture réflexive et transitive de $\rightarrow_{\mathcal{R}}$. Quand il n'y a pas d'ambiguïté sur le système de réécriture, on notera simplement $\overset{*}{\rightarrow}$.
On notera aussi $\overset{+}{\rightarrow}_{\mathcal{R}}$ la fermeture transitive de $\rightarrow_{\mathcal{R}}$, c'est-à-dire l'application successive de la relation $\rightarrow_{\mathcal{R}}$ au moins une fois. L'application successive d'une relation sur un terme peut parfois mener à une forme, dite normale, où l'on ne peut plus appliquer aucune règle.

Définition 2.3.4 *Soit $\rightarrow_{\mathcal{R}}$ la relation associée au système de réécriture $\mathcal{R}$. Un élément $e \in T_\Sigma(V)$ est une* **forme normale** *pour la relation $\rightarrow_{\mathcal{R}}$, si et seulement si il n'existe aucun autre élément $e' \in T_\Sigma(V)$ tel que $e \rightarrow_{\mathcal{R}} e'$.*
e' est appelé **forme normale** *d'un terme $e \in T_\Sigma(V)$ si et seulement si $e \overset{*}{\rightarrow}_{\mathcal{R}} e'$ et e' est une forme normale pour $\rightarrow_{\mathcal{R}}$.*

Exemple 2.3.3 *Dans l'exemple 2.3.2, le terme $succ(succ(succ(0)))$ est une forme normale pour la relation associée au système de réécriture $\mathcal{R}_{Ack}$. Aucune règle ne peut plus s'appliquer.*
C'est également une forme normale pour le terme $Ack(succ(0), succ(0))$.
$Ack(succ(0), succ(0)) \overset{*}{\rightarrow}_{\mathcal{R}_{Ack}} succ(succ(succ(0)))$

2.3.2.1 Terminaison

L'application répétitive de ces règles de réécriture pose un problème bien connu en informatique : celui de la terminaison. Un système de réécriture qui termine est simplement un système qui ne permet pas d'effectuer de dérivation infinie, c'est-à-dire qu'il n'est pas possible d'appliquer des règles indéfiniment.

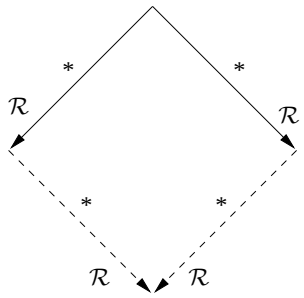
Définition 2.3.5 (Terminaison d'un système de réécriture)
On dit qu'une relation $\rightarrow_{\mathcal{R}}$ d'un système de réécriture $\mathcal{R}$ **termine** *s'il n'existe pas de* dérivation infinie $a_0 \rightarrow a_1 \rightarrow \ldots$ *lors de l'application de la relation. On dit aussi que $\overset{*}{\rightarrow}_{\mathcal{R}}$ est bien-fondée.*

Le système de réécriture pour la fonction d'Ackermann est un exemple de système de réécriture qui termine. Cependant la preuve de la terminaison n'est pas simple, et utilise les ordres récursifs (ou lexicographiques) sur les chemins que nous allons étudier dans la section 2.4.

2.3.2.2 Confluence

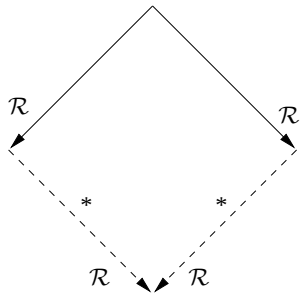
Comme nous l'avons déjà expliqué en introduction à ce chapitre, la réécriture est utilisée comme sémantique opérationnelle de la logique équationnelle. Une condition naturelle à imposer sur la relation de réécriture $\rightarrow_{\mathcal{R}}$ est qu'elle soit fonctionnelle, c'est-à-dire que sa fermeture réflexive et transitive $\overset{*}{\rightarrow}_{\mathcal{R}}$ ne puisse pas mener à deux réductions différentes à partir d'un même terme. On dit alors qu'elle est *confluente*. Ceci se définit de la façon suivante :

Définition 2.3.6 *Un système de réécriture $\mathcal{R}$ est* **confluent** *si et seulement si* $\forall y_1, y_2, x, (y_1 \; _{\mathcal{R}}\overset{*}{\leftarrow} x \overset{*}{\rightarrow}_{\mathcal{R}} y_2 \Rightarrow \exists z, y_1 \overset{*}{\rightarrow}_{\mathcal{R}} z \; _{\mathcal{R}}\overset{*}{\leftarrow} y_2)$.



La notion de confluence est une propriété difficile à montrer pour un système de réécriture. Souvent, on s'intéresse à une propriété plus faible, la confluence locale. Celle-ci nous dit que si un terme peut se réécrire en deux termes différents par *une seule étape* de réécriture, alors il existe pour chacun de ses termes une forme normale par application d'un certain nombre d'étapes de réécriture.

Définition 2.3.7 *Un système de réécriture $\mathcal{R}$ est* **localement confluent** *si et seulement si* $\forall x, y_1, y_2, (y_1 \ _{\mathcal{R}} \leftarrow x \rightarrow_{\mathcal{R}} y_2 \Rightarrow \exists z, y_1 \xrightarrow{*}_{\mathcal{R}} z \ _{\mathcal{R}} \xleftarrow{*} y_2)$.



Ceci nous amène naturellement à présenter un résultat très important en réécriture : le lemme de Newman [New42], également appelé lemme du diamant. Il permet de dire que ces propriétés de confluence et de locale confluence sont équivalentes pour tout système de réécriture qui termine.

Lemme 2.3.1 (Lemme du diamant) *Soit $\mathcal{R}$ un système de réécriture qui termine. $\mathcal{R}$ est localement confluent si et seulement si $\mathcal{R}$ est confluent.*

Une preuve originale de ce lemme sera donnée dans la section 3.3.2 page 79 de ce manuscrit.

2.4 Terminaison et ordres de simplification

Comme nous l'avons déjà vu précédemment, la réécriture est utilisée comme sémantique opérationnelle de la logique équationnelle. Une propriété importante à imposer au système de réécriture, en plus de la confluence, est alors la terminaison. Dans ce cas là, il est facile de montrer que tout terme possède nécessairement une unique forme normale pour $\overset{*}{\rightarrow}_{\mathcal{R}}$. Ceci permet alors d'utiliser le système de réécriture $\mathcal{R}$ possédant les propriétés de confluence et de terminaison comme un programme informatique. Le problème est de montrer la terminaison d'un système de réécriture ; c'est un problème extrêmement difficile.

Au lieu de montrer la terminaison de la relation $\rightarrow_{\mathcal{R}}$, une stratégie communément utilisée est de montrer que $\rightarrow_{\mathcal{R}}$ est incluse dans une autre relation binaire qui possède déjà cette propriété de terminaison. En effet, une dérivation pour un système de réécriture peut être vu comme une suite de relation binaire entre éléments de même type. Par exemple la transformation, issue de l'arithmétique $2 + (2 + 1) \rightarrow 2 + 3 \rightarrow 5$ s'intègre parfaitement dans une relation binaire $2 + (2 + 1) \succ 2 + 3 \succ 5$ où $a \succ b$ si et seulement si le nombre de symboles utilisés dans a est strictement supérieur au nombre de symboles utilisés dans b. La transformation termine si l'ordre considéré est bien-fondé.

Pour faciliter cette démarche, des classes particulières d'ordres bien fondés ont été définies. Nous allons désormais présenter une telle classe d'ordres bien-fondés : les *ordres récursifs sur les chemins*. Rappelons tout d'abord quelques notions préliminaires de base.

2.4.1 Préliminaires

2.4.1.1 Les relations binaires et les ordres

Une relation d'ordre est une relation binaire particulière. Elle s'applique sur un ensemble et permet de comparer les éléments entre eux de manière cohérente. Un ordre est un ensemble muni d'une relation d'ordre vérifiant certaines propriétés.

Exemple 2.4.1 *L'ensemble des entiers naturels $\mathbb{N}$ muni de la relation d'ordre $\leq$ classique est un ordre.*

Rappelons tout d'abord quelques propriétés bien connues des relations binaires :

Définition 2.4.1 *Soit E un ensemble. Une relation binaire r sur E est dite :*
réflexive *si* $\forall x \in E, xrx$,
symétrique *si* $\forall x, y \in E, xry \Rightarrow yrx$,
antisymétrique *si* $\forall x, y, z \in E, xry \ \wedge \ yrx \Rightarrow x = y$,

transitive *si* $\forall x, y, z \in E, xry \ \wedge \ yrz \Rightarrow xrz$.

Un ordre n'est rien d'autre qu'une relation binaire particulière sur un ensemble d'éléments.

Définition 2.4.2 *Un* **ordre** *sur E est une relation binaire sur E (notée $\succeq$ ou $\preceq$) qui est réflexive,* antisymétrique *et transitive.*

Définition 2.4.3 *Un* **pré-ordre** *sur E est une relation binaire qui est réflexive et transitive.*

Une **relation d'équivalence** *sur E est une relation binaire sur E qui est réflexive,* symétrique *et transitive.*

La **relation d'équivalence $=_\succeq$ associée à un ordre** $\succeq$ *sur E est définie par : $\forall x, y \in E, x =_\succeq y$ si et seulement si $x \succeq y$ et $y \succeq x$.*

*L'***ordre strict** $\succ$ *associé à un ordre $\succeq$ est défini par : $\forall x, y \in E, x \succ y$ si et seulement si $x \succeq y$ et $x \neq_\succeq y$.*

2.4.1.2 Relations bien-fondées

Une relation qui termine est aussi dite relation bien-fondée ou en encore nœthérienne[2]. C'est exactement une relation de ce type qui nous intéresse pour prouver la terminaison, puisqu'elle ne possèdent pas de dérivation infinie. Rappelons sa définition :

Définition 2.4.4 (Relation et ensemble nœthériens) *Soit E un ensemble. Une relation $\succcurlyeq$ sur E est dite* **nœthérienne** *(ou encore* **bien-fondée***) si et seulement si toute chaîne d'éléments $(x_i)_{i\in\mathbb{N}}$ de E liée par $\succcurlyeq$ ($x_1 \succcurlyeq x_2 \succcurlyeq \ldots$) est stationnaire, c'est-à-dire qu'il existe un $n \in \mathbb{N}$ tel que pour tout $i > n$, $x_i = x_n$. L'ensemble $(E, \succcurlyeq)$ est lui aussi dit nœthérien.*

La preuve de terminaison n'est pas décidable dans le cas général. Le résultat suivant nous assure alors qu'il est possible de prouver la terminaison d'un système de réécriture $\mathcal{R}$ en montrant que $\xrightarrow{*}_\mathcal{R}$ est incluse dans une relation bien-fondée.

Proposition 2.4.1 *Soient r_1 une relation binaire sur E et r_2 une relation nœthérienne sur E. Si on a $r_1 \subseteq r_2$, alors r_1 est aussi nœthérienne.*

Preuve Supposons qu'il existe une séquence décroissante infinie et non-stationnaire pour r_1.

$$s_1 \; r_1 \ldots \; r_1 \; s_k \; r_1 \ldots$$

[2] d'Emmy Nœther, mathématicienne allemande

Comme on a par définition $r_1 \subseteq r_2$ alors cela implique donc qu'il existe une séquence décroissante infinie non-stationnaire pour r_2

$$s_1\ r_2 \ldots\ r_2\ s_k\ r_2 \ldots$$

Ce qui est une contradiction puisque la relation r_2 est nœthérienne.

★

L'ordre nœthérien le plus simple est la relation $>$ («est supérieur à») sur les entiers naturels. Mais l'inclusion dans cet ordre, pour prouver la terminaison, n'est pas toujours facile à trouver. Par exemple, définissons la relation $\rightarrow$ sur $\mathbb{N} \times \mathbb{N}$ telle que : $(i, j+1) \rightarrow (i, j)$ et $(i+1, j) \rightarrow (i, i)$. La fonction $f : (i, j) \mapsto i^2 + j$ permet l'inclusion de $\overset{*}{\rightarrow}_{\mathcal{R}}$ dans $>$, mais elle ne vient pas à l'esprit immédiatement !

Lorsque les ordres habituels ne suffisent pas, il est possible d'en créer de nouveaux par combinaison des premiers. C'est ce que nous allons voir dans la section 2.4.2 avec les *ordres récursifs sur les chemins*.

2.4.1.3 Les multi-ensembles

Dans cette section, nous allons décrire une structure intermédiaire entre les ensembles et les listes qui rentre en jeu dans la définition des ordres récursifs sur les chemins.

Dans un ensemble, les éléments n'apparaissent qu'une seule fois. Il est souvent très utile d'avoir des ensembles dans lesquels les éléments peuvent apparaître plusieurs fois. Les multi-ensembles permettent ceci. Il se comportent comme des listes non ordonnées où les répétitions sont autorisées. Ils sont très utiles notamment en réécriture, et pour ce qui nous concerne pour prouver des résultats de terminaison tels que nous les verrons dans le chapitre 3. Ils permettent aussi de définir une classe particulière du calcul des séquents où certaines règles structurelles sont explicites (voir chapitre 3).

Présentation

Définition 2.4.5 *Un* **multi-ensemble** *est une paire (E, m) où E est un ensemble et $m : E \rightarrow \mathbb{N}$ une fonction qui à tout élément $e \in E$, associe un nombre entier positif qui représente le nombre d'occurrences de cet élément dans le multi-ensemble.*

En particulier, un élément $a \in E$ n'appartient pas au multi-ensemble (E, m) si et seulement si $m(a) = 0$. On notera $M(E)$ l'ensemble de tous les multi-ensembles construits à partit de l'ensemble E.

On peut écrire les multi-ensembles de manières différentes :

- comme un ensemble de couples $\bigcup_{e \in E}(e, m(e))$
- comme une séquence d'éléments entre double-accolades $\{\{x_1, x_2, \ldots, x_n\}\}$

Exemple 2.4.2 *$\{\{x, x, y, z, z, z\}\}$ et $\{(x, 2), (y, 1), (z, 3)\}$ dénotent le même multi-ensemble.*

Propriétés des multi-ensembles Les propriétés des multi-ensembles sont très proches de celles des ensembles, mais nécessitent des définitions adaptées que nous donnons dans cette section.

Définition 2.4.6 *La* **taille** *d'un multi-ensemble (E, m), notée $|(E, m)|$, est la somme de toutes les occurrences des éléments de E donnée par la fonction m :*

$$|(E, m)| = \Sigma_{e \in E} m(e)$$

Le multi-ensemble $C = \{\{x, x, y, z, z, z\}\}$ est de taille $|C| = 2 + 1 + 3 = 6$.

Définition 2.4.7 *Un multi-ensemble (E', n) est un* **sous multi-ensemble** *de (E, m), noté $(E', n) \subseteq (E, m)$, si et seulement si E' est un sous-ensemble de E, $E' \subseteq E$, et pour tout élément $e \in E'$, $n(e) \leq m(e)$.*

$\{\{x, y, z\}\}$ est un sous multi-ensemble de C, mais $\{\{x, y, y, z\}\}$ n'en est pas un car y apparaît deux fois alors qu'il n'apparaît qu'une fois dans C.

On peut redéfinir les opérations habituelles des ensembles, comme la différence et l'union de deux multi-ensembles.

Définition 2.4.8 *Soit (E, m) et (E, n) des multi-ensembles.*

*L'***union** *des deux multi-ensembles, notée $(E, m) \cup (E, n)$, est définie par la fonction associée $f : E \to \mathbb{N}$ telle que, pour tout $e \in E$, $f(e) = m(e) + n(e)$. On note (E, f) le multi-ensemble résultant de l'union.*

La **différence** *des deux multi-ensembles, notée $(E, m) - (E, n)$, est le multi-ensemble (E, f) tel que la fonction associée $f : E \to \mathbb{N}$ est définie de la manière suivante : pour tout $x \in E$,*
$f(x) = n(x) - m(x)$ si $n(x) \geq m(x)$,
$f(x) = 0$ sinon.

Ordres et multi-ensembles Comme pour les ensembles, il est possible de définir des relations d'ordre sur les multi-ensembles. La comparaison de deux multi-ensembles (E, n) et (E', m), se fait via une relation d'ordre sur les multi-ensembles notée $\prec\!\!\prec$ obtenue à partir d'une relation $\preceq$ sur les ensembles. Pour pouvoir dire qu'un des multi-ensembles est plus petit que l'autre, il faut que (E, n) soit obtenu à partir de (E', m) en retirant zéro ou plusieurs éléments de (E', m), et en remplaçant (ou non) chacun de ces éléments x par des éléments plus petit que x (dans l'ordre associé $\preceq$).

Exemple 2.4.3 *Par exemple, pour une relation d'ordre classique $\leq$ sur l'ensemble des entiers naturels $\mathbb{N}$, on aura l'ordre $\ll$ sur $M(\mathbb{N})$. Ainsi on aura :*
$\{\{2, 3, 5, 5\}\} \ll \{\{2, 2, 3, 5, 5, 5\}\}$
$\{\{1, 2, 3, 4, 5\}\} \ll \{\{2, 2, 3, 5, 5, 5\}\}$
mais on aura pas :
$\{\{2, 3, 5, 6\}\} \ll \{\{2, 3, 5, 5\}\}$.

Voici une définition plus formelle :

Définition 2.4.9 *Soit $\preceq$ un ordre associé à un ensemble E, la relation $\prec\!\!\prec$ sur l'ensemble des multi-ensembles $M(E)$.*

Soient (E, n) et (E', m) deux multi-ensembles, $(E, n) \prec\!\!\prec (E', m)$ si et seulement si :

- *$(E, n) = (E', m)$, ou*
- *il existe des multi-ensembles finis $(X, g), (Y, h)$, avec $(X, g) \subseteq (E', m)$, tels que $(E, n) = ((E', m) - (X, g)) \cup (Y, h)$, et pour tout $y \in Y$ il existe $x \in X$ tel que $y \prec x$*

2.4.2 Ordres récursifs sur les chemins

Voici donc, la présentation des ordres récursifs (ou lexicographiques) sur les chemins. Ils nous permettrons également de prouver la terminaison de nos procédures de normalisation d'arbres de preuve dans le chapitre 3.

2.4.2.1 Ordres de réduction

Nous voulons prouver la terminaison d'un système de réécriture défini sur la base d'un ensemble $\mathcal{R}$ de schémas de règles. L'instanciation de ces règles par application de substitutions et de contextes permet d'obtenir la relation notée $\rightarrow_{\mathcal{R}}$ (voir définition 2.3.2 page 48). Nous avons envie de prouver l'inclusion dans un ordre nœthérien $\succcurlyeq$ de touts les schémas de $\mathcal{R}$. Comme on l'a vu dans la définition 2.3.2 de la section sur la réécriture, une étape de réécriture implique que la relation

soit stable par substitution et par passage au contexte. Pour garantir la terminaison de la relation complète, il faut donc que $\succcurlyeq$ soit stable par passage au contexte et par substitution. La relation $\succcurlyeq$ est alors appelée ordre de réduction.

Définition 2.4.10 *Un* **ordre de réécriture** $\succcurlyeq$ *est un ordre clos par substitution et par passage au contexte : si* $t \succcurlyeq t'$ *alors pour toute substitution* σ *et tout contexte* $s[.]_w$, $s[\sigma t]_w \succcurlyeq s[\sigma t']_w$.
Un **ordre de réduction** *est un ordre de réécriture nœthérien.*

Nous pouvons ainsi introduire le résultat suivant, dont un des sens est une spécialisation de la proposition 2.4.1 page 52 :

Théorème 2.4.1 (de Lankford) *Un système de réécriture de terme* $\rightarrow_{\mathcal{R}}$ *termine si et seulement s'il existe un* ordre de réduction $\succcurlyeq$ *tel que* $\mathcal{R} \subseteq \succcurlyeq$ *(c'est-à-dire que* $l \succcurlyeq r$ *pour toute règle* $l \rightarrow r \in \mathcal{R}$*).*

Preuve

1. ($\Rightarrow$) On suppose que la relation $\mathcal{R}$ termine. Pour toutes expressions s, t telles que $s \rightarrow_{\mathcal{R}} t$, il est évident qu'il existe un ordre nœthérien $>$ tel que $s > t$. La relation ne terminerait pas sinon. On peut donc affirmer que $>$ est un ordre de réduction puisqu'une relation de réécriture est par définition close par substitution et par contexte.
2. ($\Leftarrow$) Inversement, si il existe un ordre de réduction $>$ tel que pour toute règle $l \rightarrow r \in \mathcal{R}$ on a $l > r$, quelque soit la séquence décroissante $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \ldots$ correspondante à la séquence décroissante $t_1 > t_2 > t_3 > \ldots$ celle-ci termine. En fait, si ce n'était pas le cas, cela signifierait que l'ordre de réduction $>$ n'est pas nœthérien.

★

2.4.2.2 Ordres de simplification

Les ordres récursifs sur les chemins font partie d'une classe plus grande d'ordres de réduction sur les termes : les ordres de simplification.

Définition 2.4.11 (Ordre de simplification) *Soit* $\Sigma = (S, F, V)$ *une signature. Un ordre* $\succ$ *sur* $T_\Sigma(V)$ *est un ordre de simplification si pour chaque nom de fonction* f *d'arité* $n \in \mathbb{N}$*, on a*

1. *la stabilité par contexte : si pour tout* $1 \leq i \leq n, t_i \succ t'_i$*, alors* $f(t_1, \ldots, t_n) \succ f(t'_1, \ldots, t'_n)$.
2. *la propriété de sous-terme : pour tout* $1 \leq i \leq n$*,* $f(t_1, \ldots, t_n) \succ t_i$.

Ces ordres sont bien-fondés, c'est ce que nous dit le théorème de Dershowitz en se fondant sur un résultat plus ancien, le théorème de Kruskal. La preuve classique, que l'on peut trouver dans [Lal90, BN98] s'appuie sur le la notion de bon pré-ordre, et sur une relation dite de plongement.

Définition 2.4.12 (Relation de plongement) *Soit $\Sigma = (S, F, V)$ une signature. La relation $\sqsubseteq$ de* **plongement** *est définie par :*

$$t = f(t_1, \ldots, t_n) \sqsubseteq g(t'_1, \ldots, t'_m) = t'$$

avec t, t', t_i, t'_j des termes, si l'une des trois conditions suivantes est satisfaite :

1. $t = t'$,
2. $f = g$ *et* $t_i \sqsubseteq t'_i$, *pour tout* i, $1 \leq i \leq p$,
3. *il existe* i, $1 \leq i \leq m$ *tel que* $t \sqsubseteq t'_i$.

Ainsi, $t \sqsubseteq t'$ si t peut être obtenu à partir de t' en "élaguant" t'. Plaçons nous dans le cadre plus général des pré-ordre.

Définition 2.4.13 *Un pré-ordre $\leq$ est un* **bon pré-ordre** *si pour toute séquence infinie $(x_n)_{n\in\mathbb{N}}$ il existe i, j tel que $i < j$ et $x_i \leq x_j$.*

L'intérêt des bon pré-ordres est qu'ils sont nécessairement bien fondés. Le théorème de Kruskal nous permet alors de prouver que l'ordre de plongement est bien un bon pré-ordre et donc qu'il est bien fondé. En voici l'énoncé :

Théorème 2.4.2 (Kruskal) *Soit Σ une signature finie. La relation de plongement $\sqsubseteq$ est un bon pré-ordre sur les termes clos T_Σ.*

Comme nous l'avons déjà dit la preuve de ce théorème est classique, et on peut la trouver dans [Lal90, BN98]. Il nous permet de montrer le résultat suivant :

Théorème 2.4.3 *Les ordres de simplification sont bien fondés.*

Ce résultat nous permettra ainsi de prouver la terminaison de certains systèmes de réécriture lorsque ceux-ci sont contenus dans un ordre de simplification. C'est le théorème de Dershowitz [Der82, DJ90] qui nous l'assure :

Théorème 2.4.4 (Théorème de Dershowitz) *Un système de réécriture $\mathcal{R}$ est bien-fondé s'il existe un ordre de simplification $\succ$ tel que $\sigma(t) \succ \sigma(t')$ pour toute règle $t \to t' \in \mathcal{R}$ et pour toute substitution σ.*

2.4.2.3 Ordres récursifs sur les chemins

L'idée sous-jacente aux ordres récursifs sur les chemins (*recursive path orderings* en anglais) est que deux termes sont ordonnés en comparant leurs symboles à la racine selon un ordre nœthérien sur les symboles de fonctions de la signature et, de façon récursive, les multi-ensembles de leurs sous-termes immédiats.

Définition 2.4.14 (Ordre récursif sur les chemins) *Soient Σ une signature et V un ensemble de variables. Soit $\succ$ un ordre "de précédence" sur $\Sigma \cup V$ tel que deux variables différentes sont incomparables, et les symboles de fonction et les variables sont incomparables. On construit à partir de $\succ$ l'ordre récursif sur les chemins (*rpo *ou* recursive path ordering*, en anglais) $\succ_{rpo}$ sur l'ensemble des termes $T_\Sigma(V)$ et l'ordre associé $\succcurlyeq_{rpo}$ sur $\mathcal{M}(T_\Sigma(V))$ l'ensemble des multi-ensembles de termes.*

On note $\sim_{rpo}$ l'équivalence par permutation des sous-termes : $f(t_1, \ldots, t_n) \sim_{rpo} f(t'_1, \ldots, t'_n)$ si $t_i \sim_{rpo} t'_{\pi(i)}$ pour une permutation des indices allant de 1 à n ; et $t \succcurlyeq_{rpo} t'$ si $t \succ_{rpo} t'$ ou $t \sim_{rpo} t'$.

$$t = f(t_1, \ldots, t_n) \succ_{rpo} g(t'_1, \ldots, t'_m) = t'$$

si l'une des trois conditions suivantes est satisfaite :

(RPO1) $f = g$ (donc $m = n$), et $\{\!\{t_1, \ldots, t_n\}\!\} \succcurlyeq_{rpo} \{\!\{t'_1, \ldots, t'_n\}\!\}$
(RPO2) $f \succ g$ et pour tout $i, 1 \leq i \leq m, t \succ_{rpo} t'_i$
(RPO3) $f \not\succcurlyeq g$ et il existe $i, 1 \leq i \leq n$ tel que $t_i \succcurlyeq_{rpo} t'$

Les ordres récursifs sur les chemins sont des ordres de simplification, donc ils sont également bien-fondés [Lal90]. Nous allons pouvoir les utiliser pour prouver la terminaison de systèmes de réécriture.

2.4.2.4 Ordres lexicographiques sur les chemins

Un ordre lexicographique est un ordre récursif (*lexicographic path order*) sur les chemins particulier où les multi-ensembles sont des n-uplets.

Il faut d'abord définir l'extension lexicographique de n relations d'ordre. De plusieurs relations d'ordres élémentaires on pourra ainsi définir un nouvel ordre construit lexicographiquement.

Définition 2.4.15 (Extension lexicographique) *Soient $(E_1, \succcurlyeq_1), \ldots, (E_n, \succcurlyeq_n)$, n ensembles ordonnés. L'* **extension lexicographique** *de $\succcurlyeq_1, \ldots, \succcurlyeq_n$, notée $\succcurlyeq_{lex}$, est définie sur $E_1 \times \ldots \times E_n$ par :*

$$(x_1, \ldots, x_n) \succcurlyeq_{lex} (y_1, \ldots, y_n)$$

si et seulement si on a :

- $\forall i(1 \leqslant i \leqslant n),\ x_i = y_i$, *ou bien*
- $\exists i(1 \leqslant i \leqslant n),\ x_i \succcurlyeq_i y_i \wedge \forall j(1 \leq j < i),\ x_j = y_j$

Proposition 2.4.2 *Si les ensembles ordonnés* $(E_1, \succcurlyeq_1), \ldots, (E_n, \succcurlyeq_n)$ *sont tous nœthériens, alors leur extension lexicographique* $\succcurlyeq_{lex}$ *est elle aussi nœthériene.*

Preuve Par l'absurde, supposons qu'il existe une séquence décroissante infinie non-stationnaire pour $\succcurlyeq_{lex}$.

$$(s_1, \ldots, s_n) \succcurlyeq_{lex} \ldots \succcurlyeq_{lex} (t_1, \ldots, t_n) \succcurlyeq_{lex} \ldots$$

Par définition de $\succcurlyeq_{lex}$ on sait qu'il existe un indice i tel que l'on ait une suite décroissante infinie non-stationnaire telle que :

$$s_i \succcurlyeq_i \ldots \succcurlyeq_i t_i \succcurlyeq_i \ldots$$

Ceci est une contradiction puisque l'on a supposé que tous les ensembles étaient munis d'une relation nœthérienne. ★

Sur le même principe, il est possible d'étendre la définition de l'extension lexicographique pour comparer des séquences bornées (mais de tailles différentes) d'éléments de plusieurs ensembles.

Définition 2.4.16 (Ordre lexicographique) *Soient* $(E_1, \succcurlyeq_1), \ldots, (E_n, \succcurlyeq_n)$, n *ensembles ordonnés.* **L'ordre lexicographique**, *noté* $\succcurlyeq_{lexe}$, *est défini de la manière suivante :*

$$\forall m, p \leq n,\ \ (s_1, \ldots, s_m) \succcurlyeq_{lexe} (t_1, \ldots, t_p) \Leftrightarrow (s_1, \ldots, s_k) \succcurlyeq_{lex} (t_1, \ldots, t_k),$$

k *étant le minimum entre* m *et* p.

Proposition 2.4.3 *Si les ensembles ordonnés* $(E_1, \succcurlyeq_1), \ldots, (E_n, \succcurlyeq_n)$ *sont tous nœthériens, alors* **l'ordre lexicographique** $\succcurlyeq_{lexe}$ *est lui aussi nœthérien.*

Preuve Par l'absurde, supposons qu'il existe une séquence décroissante infinie non-stationnaire pour $\succcurlyeq_{lexe}$.

$$(s_1, \ldots, s_m) \succcurlyeq_{lexe} \ldots \succcurlyeq_{lexe} (t_1, \ldots, t_p) \succcurlyeq_{lexe} \ldots$$

L'ordre de cette séquence s'établit uniquement sur les n premiers éléments des tuples, sachant que n représente la longueur du tuple de plus petite taille de la séquence. On pose k comme étant la longueur du tuple de plus grande taille de la séquence. Il existe donc un tuple de termes quelconque de taille k, $(s'_1, \ldots, s'_k)$,

tel que l'on puisse compléter tous les tuples de taille strictement inférieure à k par les $k - l$ derniers éléments de cet ensemble, l étant la taille du tuple à compléter. On obtient ainsi une séquence infinie ordonnée selon $\succcurlyeq_{lex}$:

$$(s_1, \ldots, s_m, s'_{m+1}, \ldots, s'_k) \succcurlyeq_{lex} \ldots \succcurlyeq_{lex} (t_1, \ldots, t_p, s'_{p+1}, \ldots, s'_k) \succcurlyeq_{lex} \ldots$$

On en arrive donc à une contradiction puisque *l'extension lexicographique* $\succcurlyeq_{lex}$ est nœthérienne. ★

L'ordre lexicographique sur les chemins va nous permettre de prouver la terminaison de systèmes de réécriture, notamment dans l'exemple 2.3.1 page 47 de la fonction d'Ackermann.

Définition 2.4.17 (Ordre lexicographique sur les chemins) *Soient Σ une signature, V un ensemble de variables, et $\geqslant_F$ un ordre nœthérien sur F (les symboles de fonctions).* **L'ordre lexicographique sur les chemins** $>^{lpo}$ *est défini sur les termes de $T_\Sigma(V)$ par $s >^{lpo} t$ si :*

1. **(LPO1)** $t \in Var(s)$ *et* $s \neq t$ *ou bien*
2. **(LPO2)** $s = f(s_1, \ldots, s_n)$ *et* $t = g(t_1, \ldots, t_m)$ *et l'une des conditions suivantes est vérifiée :*
   - *(a)* **(LPO2a)** $f =_F g$, $(s_1, \ldots, s_n) >^{lpo}_{lexe} (t_1, \ldots, t_m)$ *et* $\forall i \in \{1, \ldots, m\}$, $s >^{lpo} t_i$ *;*
   - *(b)* **(LPO2b)** $f \geqslant_F g$ *et* $\forall i \in \{1, \ldots, m\}, s >^{lpo} t_i$;
   - *(c)* **(LPO2c)** $\exists i \in \{1, \ldots, n\}$ *tel que* $s_i >^{lpo} t$ *ou bien* $s_i = t$ *(c'est-à-dire* $s_i \geqslant t$*).*

Il s'agit en fait d'un ordre récursif sur les chemins où l'on remplace la condition $(RPO1)$ par la condition $(LPO2a)$.

Théorème 2.4.5 *Les ordres lexicographiques sur les chemins sont des ordres de simplification.*

La preuve est similaire à celle faite pour les ordres récursifs sur les chemins et pour les lecteurs intéressés, est disponible dans la littérature traitant de ce sujet, par exemple [BN98, Lal90].

2.4.2.5 Exemple : La fonction d'Ackermann

À partir du système de réécriture $\mathcal{R}$ défini dans l'exemple 2.3.1 page 47, que l'on rappelle ici, nous allons montrer que la relation $\xrightarrow{*}_{\mathcal{R}}$ est incluse dans un ordre lexicographique sur les chemins et donc que le système de réécriture termine.

$$\begin{array}{lrcl}
(1) & Ack(0, y) & \rightarrow & succ(0) \\
(2) & Ack(succ(x), 0) & \rightarrow & Ack(x, succ(0)) \\
(3) & Ack(succ(x), succ(y)) & \rightarrow & Ack(x, Ack(succ(x), y))
\end{array}$$

Commençons d'abord par définir l'ordre bien-fondé $>_F$ sur les symboles de fonctions :

$$Ack >_F succ >_F 0$$

Montrons alors que la relation $\rightarrow$ vérifie l'ordre lexicographique sur les chemins $>^{lpo}$ sous-jacent à $>_F$. Nous devrons alors montrer que chacune des règles de la fonction d'Ackermann est incluse dans $>^{lpo}$. Par fermeture par contexte, substitution et transitivité, nous aurons alors nécessairement $\xrightarrow{*}_{\mathcal{R}} \subseteq >^{lpo}$.

- Pour l'équation (1) de la fonction d'Ackermann, nous devons montrer que $Ack(0, x) >^{lpo} succ(0)$.
  Nous nous trouvons dans le cas **(LPO2b)**, avec $s = Ack(0, x)$ et $t = succ(0)$, donc $n = 2$ et $m = 1$. Nous savons, d'après l'ordre défini sur les symboles, que $Ack >_F succ$. Nous devons donc montrer que $\forall i \in \{1, \ldots, m\}$ (c'est-à-dire pour $i = 1$ car $m = 1$), $s >^{lpo} t_1$ (c'est-à-dire que $Ack(0, y) >^{lpo} 0$).
  Pour montrer que $Ack(0, y) >^{lpo} 0$, profitons du fait que l'ordre lexicographique sur les chemins a la propriété de sous-terme, c'est-à-dire que tout terme est plus grand que chacun de ses sous-termes. Nous avons donc immédiatement que $Ack(0, y) >^{lpo} 0$ puisque 0 est un sous-terme de $Ack(0, y)$.
  Nous avons donc bien démontré que l'équation (1) de la fonction d'Ackermann vérifie l'ordre lexicographique sur les chemins.

- Pour l'équation (2), montrons que $Ack(succ(x), 0) >^{lpo} Ack(x, succ(0))$. Nous avons $s = Ack(succ(x), 0)$, $t = Ack(x, succ(0))$, et donc $n = m = 2$.
  Nous nous trouvons dans le cas **(LPO2a)** avec $f =_F g =_F Ack$. Il faut donc prouver deux choses différentes :

  1. $(s_1, s_2) >^{lpo}_{lexe} (t_1, t_2)$ c'est-à-dire que $(succ(x), 0) >^{lpo}_{lexe} (x, succ(0))$. Nous allons en fait comparer les ensembles des arguments des deux côtés de l'équation selon l'*extension lexicographique des ordres* (voir définition 2.4.15).

     Nous avons $m = n = 2$ (la cardinalité des tuples à comparer) et nous devons montrer $(s_1, s_2) >^{lpo}_{lex} (t_1, t_2)$ car les deux ensembles sont de même cardinalité.

     Il nous suffit donc simplement de prouver que $succ(x) >^{lpo} x$. Ce qui est immédiat car x est un sous-terme de $succ(x)$.

2. $\forall i \in \{1,2\}$, $s >^{lpo} t_i$.
   Pour $i = 1$, montrons que $Ack(succ(x),0) >^{lpo} x$, ce qui est immédiat puisque x est un sous-terme de $Ack(succ(x),0)$. Pour $i = 2$, montrons que $Ack(succ(x),0) >^{lpo} succ(0)$. Comme pour l'équation (1), nous utilisons **(LPO2b)** et nous devons donc prouver $Ack(succ(x),0) >^{lpo} 0$, ce qui est de nouveau une conséquence immédiate de la propriété des ordres de sous-terme.

– Et puis, pour l'équation (3), nous devons montrer que $Ack(succ(x),succ(y)) >^{lpo} Ack(x,Ack(succ(x),y))$. Comme pour l'équation (2), nous passons d'abord par **(LPO2a)**,

1. Pour le premier cas nous sommes dans la même situation.
2. Pour le deuxième c'est un peu différent.
   Pour $i = 1$, montrons que $Ack(succ(x),succ(y)) >^{lpo} x$. C'est le même problème que dans l'équation (2), nous sommes dans le cas **(LPO1)**.
   Pour $i = 2$, montrons que $Ack(succ(x),succ(y)) >^{lpo} Ack(succ(x),y)$. Nous nous retrouvons dans le cas **(LPO2a)** :
   (a) $(succ(x),succ(y)) >^{lpo}_{lex} (succ(x),y)$ car $succ(x) = succ(x)$, et $succ(y) >^{lpo} y$ (selon l'ordre de sous-terme)
   (b) Pour $i = 1$, $Ack(succ(x),succ(y)) >^{lpo} succ(x)$, que nous pouvons comme dans les cas précédent montrer grâce à la propriété de sous-terme.
   Pour $i = 2$, $Ack(succ(x),succ(y)) >^{lpo} y$ car y est un sous-terme de $Ack(succ(x),succ(y))$.

Ceci termine donc l'exemple. Nous avons bien ce que nous voulions démontrer.

2.5 Systèmes de réécriture conditionnelle

Dans cette section, nous présentons une classe particulière de systèmes de réécriture [KK06], les systèmes de réécriture conditionnelle. Les formules considérées sont alors des *formules conditionnelles* de la forme

$$t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow l = r$$

où $t_1 = t'_1, \ldots, t_n = t'_n$ sont les *conditions*, et $l = r$ la *conclusion* de la formule.

Les *règles de réécriture conditionnelle* sont construites à partir de formules conditionnelles dont on oriente la conclusion et on les note

$$t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow l \rightarrow r$$

Définition 2.5.1 *Un* **système de réécriture conditionnelle** *est un ensemble de règles de réécriture conditionnelle de la forme*

$$t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow l \rightarrow r$$

où $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$ sont les conditions, et l et r sont, respectivement, les parties gauches et droites des conclusions des règles de réécriture conditionnelle.

Pour un système de réécriture $\mathcal{R}$, une règle donnée s'applique sur un terme si ses conditions sont satisfaites lorsqu'elles sont instanciées par la substitution qui convient.

Dans les systèmes de réécriture classiques, les variables, contenues dans le membre droit d'une règle, apparaissent également dans le membre gauche. On généralise cette condition aux systèmes de réécriture conditionnelle : les variables contenues dans les conditions ou dans le membre droit, apparaissent également dans le membre gauche de la règle. Cette condition n'est pas nécessaire dans le cas général, mais est bien souvent requise.

Définition 2.5.2 *Soit $\mathcal{R}$ un système de réécriture conditionnelle. Un terme t* **se réécrit** *en un terme t', que l'on note $t \rightarrow_{\mathcal{R}} t'$ si il existe :*

1. *une règle de réécriture $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow l \rightarrow r$ dans $\mathcal{R}$,*
2. *une position ω dans le terme t,*
3. *une substitution σ, satisfaisant $t|_\omega = \sigma(l)$,*
4. *une substitution τ telle que les conditions $\tau(\sigma(t_1)) = \tau(\sigma(t'_1)) \wedge \cdots \wedge \tau(\sigma(t_n)) = \tau(\sigma(t'_n))$ soient vérifiées.*

On a ainsi $t' = t[\tau(\sigma(r))]_\omega$.

D'après cette définition, une règle de réécriture conditionnelle est applicable si il existe une substitution sur les variables telle que les conditions soient satisfaites. Ce problème peut être résolu par un processus appelé "narrowing" en anglais. Mais la définition 2.5.2 est trop générale pour être applicable. Il faut donc poser des conditions plus restrictives. La première chose requise est que chaque variable apparaissant soit dans les conditions, soit dans la partie droite de la conclusion doit aussi apparaître dans la partie gauche de la conclusion, c'est-à-dire

$$Var(t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n) \cup Var(r) \subseteq Var(l)$$

Puis dans la condition 4, τ devient l'identité, et nous la remplaçons par la condition suivante :

4. *pour tout* i, $\sigma(t'_i)$ *est une forme normale de* $\sigma(t_i)$

La propriété suivante de réduction sur les systèmes de réécriture conditionnelle nous permettra de garantir des propriétés pour un de nos critères de sélection donné dans le chapitre 6 de ce manuscrit.

Définition 2.5.3 (Réduction) *Un système de réécriture conditionnelle* $\mathcal{R}$ *est* **réducteur** *si et seulement si il est équipé d'un ordre bien fondé qui satisfait pour toutes les règles* $\bigwedge_{1\leq i\leq m} t_i = t'_i \Rightarrow t \rightarrow t'$ *dans* $\mathcal{R}$ *et toutes les substitutions* σ *:*

- $\sigma(t) > \sigma(t')$, *et*
- $\forall i \in \{1, \ldots, m\}, \sigma(t) > \sigma(t_i), \sigma(t) > \sigma(t'_i)$.

Deuxième partie

Un résultat général de normalisation d'arbres de preuve

Chapitre 3

Normalisation d'arbres de preuve

Dans beaucoup de situations en logique, pour faciliter l'utilisation des systèmes d'inférence, ou pour obtenir des résultats de cohérence ou de décidabilité sur les systèmes de preuve, ou encore pour faire de la preuve automatique, on utilise souvent une stratégie d'application des règles d'inférence. L'intérêt est de réduire l'espace de recherche des arbres de preuve. Citons quelques exemples où une stratégie d'application des règles d'inférences peut être utilisée :

- le résultat de logicalité pour de nombreuses formes de logiques équationnelles [vO04, Bir35, Kap84, ABD02] ;
- l'élimination des coupures qui montre que la règle de coupure n'est pas nécessaire pour le calcul des séquents et pour la déduction naturelle dans de nombreuses logiques (logique du premier ordre [Gen35], logique intuitionniste [Kle52], logiques modales [Wal90], déduction modulo [DW99], display logic [DG03]) ;
- la propriété de confluence pour les systèmes de réécriture (voir la section 2.3.2.2) ;

Pour savoir si on ne perd pas en puissance de preuve en utilisant ces stratégies, il faut montrer que n'importe quel arbre de preuve, sans stratégie d'application des règles, coïncide exactement avec la classe d'arbres de preuve, correspondante à la stratégie particulière d'application des règles. Nous dirons alors qu'elles sont correctes et complètes par rapport au système d'inférence de départ.
La *correction* est trivialement satisfaite car les arbres résultants de telles stratégies sont avant tout des arbres de preuve.
La *complétude* est beaucoup plus complexe. En effet, elle demande que pour tout énoncé de théorème, il existe un arbre de preuve associé satisfaisant la structuration imposée par la stratégie d'application des règles d'inférence. Usuellement, dans tous les systèmes formels où ce type de démarche de restriction de l'espace de recherche des arbres de preuve a été appliqué, la complétude est une consé-

quence d'un résultat plus fort qui consiste à définir des transformations d'arbre de preuve élémentaires, de montrer que ces transformations terminent et que l'espace des preuves ainsi obtenues "couvre" l'espace des théorèmes. On observe alors que ces transformations se caractérisent par des transformations élémentaires qui consistent souvent à faire remonter certaines règles sur d'autres. On parle alors de *normalisation d'arbres de preuve*. En général, dans les systèmes formels où de tels résultats de normalisation ont été appliqués, la difficulté ne réside pas dans la définition de ces transformations élémentaires mais dans la preuve de terminaison de ces dernières. À chaque fois que de tels résultats de normalisation d'arbres de preuve ont été établis, c'est de façon ad-hoc au système formel sous-jacent.

Dans cette partie, nous allons montrer que de nombreux résultats peuvent s'unifier et ainsi se généraliser indépendemment du système formel sous-jacent. Ainsi, nous allons montrer que des résultats de normalisation aussi différents que la logicalité, le lemme de Newman ou encore l'élimination des coupures dans le calcul des séquents et en déduction naturelle s'unifient dans un même cadre général que nous allons définir dans cette partie. Les résultats de normalisation d'arbres de preuve présentés ici seront largement utilisés dans toute la suite de la thèse. Nous les utiliserons notamment dans les preuves du chapitre 6 pour prouver la complétude de notre méthode de dépliage.

Ce travail a été commencé pendant mon stage de DEA [Boi03], a été poursuivi par la suite pendant ma thèse, et a donné lieu a un rapport de recherche du LaMI [ABL05]. Il est inspiré d'un premier résultat de normalisation d'arbres de preuve décrit dans [Bah03, ABD02].

Nous allons tout d'abord énoncer nos résultats de normalisation d'arbre de de preuve sous la forme de résultats généraux valides pour n'importe quel système formel. Puis, nous donnerons plusieurs exemples détaillés de normalisation.

3.1 Les transformation d'arbres de preuve

3.1.1 Introduction

Dans les systèmes de preuves que nous avons étudiés précédemment, plusieurs arbres de preuve peuvent être associés à la preuve d'une même formule, tout dépend de la façon dont les règles sont appliquées. Par exemple, construisons la preuve de la formule valide $(A \wedge B) \Rightarrow (A \vee B)$ dans le calcul des séquents G vu dans la section 1.2.3.1 page 27 :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \rightarrowtail A, B}\, Ax}{(A \wedge B) \rightarrowtail A, B}\, \wedge Gauche}{(A \wedge B) \rightarrowtail (A \vee B)}\, \vee Droite}{\rightarrowtail (A \wedge B) \Rightarrow (A \vee B)}\, \Rightarrow Droite$$

Nous aurions pu l'écrire aussi bien de la manière suivante, en appliquant d'abord la règle $\wedge Gauche$ puis la règle $\vee Droite$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \rightarrowtail A, B}\, Ax}{A, B \rightarrowtail (A \vee B)}\, \vee Droite}{(A \wedge B) \rightarrowtail (A \vee B)}\, \wedge Gauche}{\rightarrowtail (A \wedge B) \Rightarrow (A \vee B)}\, \Rightarrow Droite$$

La conclusion et l'unique feuille (Ax) de l'arbre sont les mêmes dans les deux arbres. Seule la structure interne de l'arbre a changé.

Normaliser un arbre de preuve consiste alors à transformer cette structure interne de l'arbre pour lui donner une configuration particulière (pas forcément unique). Nous allons donc donner un cadre générique adapté à la normalisation d'arbre de preuve le plus indépendant possible d'un système formel donné. Nous allons normaliser via l'utilisation de transformations élémentaires d'arbres de preuve. Ces transformations fonctionnent comme des règles de réécriture (voir section 2.3 page 46). Nous pouvons, par exemple, transformer notre premier arbre de preuve en notre second arbre, grâce à la règle suivante :

$$\dfrac{\dfrac{\Gamma, A, B \rightarrowtail A, B, \Delta}{\Gamma, (A \wedge B) \rightarrowtail A, B, \Delta}\, \wedge Gauche}{\Gamma, (A \wedge B) \rightarrowtail (A \vee B), \Delta}\, \vee Droite \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma, A, B \rightarrowtail A, B, \Delta}{\Gamma, A, B \rightarrowtail (A \vee B), \Delta}\, \vee Droite}{\Gamma, (A \wedge B) \rightarrowtail (A \vee B), \Delta}\, \wedge Gauche$$

Un telle règle permet de dire que $\vee Droite$ "remonte" sur $\wedge Gauche$. Quand on l'applique sur un arbre de preuve, et que la partie à gauche du signe $\rightsquigarrow$ s'unifie avec une partie élémentaire de l'arbre, alors on réécrit cette partie élémentaire par la partie droite. On peut le faire car les conclusions et les feuilles sont les mêmes des deux côtés.

Pour fonder notre résultat de normalisation d'arbres de preuve nous utilisons les notions de réécriture et de terminaison que nous avons déjà présentées dans la section 2.3 page 46. Pour normaliser un arbre de preuve, nous utilisons des règles de transformation élémentaires. Quand elles sont regroupées, elles forment des systèmes de transformation d'arbres de preuve. Il nous faudra alors définir des conditions sur ces systèmes de transformation pour garantir leur terminaison.

3.1.2 Arbres de preuve élémentaires

La normalisation va donc se faire pas à pas en transformant un arbre de preuve complet par des transformations de "petits morceaux" ou motifs internes de l'arbre. Ces motifs élémentaires sont appelés *arbres de preuve élémentaires* et sont constitués d'une application de règle, dont les prémisses de celle-ci sont soit des feuilles soit des applications de règles.

Par exemple, la transformation donnée en début de chapitre

$$\dfrac{\dfrac{\Gamma, A, B \rightarrowtail A, B, \Delta}{\Gamma, (A \wedge B) \rightarrowtail A, B, \Delta}\ \wedge Gauche}{\Gamma, (A \wedge B) \rightarrowtail (A \vee B), \Delta}\ \vee Droite$$

est un arbre de preuve élémentaire. Donnons la définition formelle de tels arbres de preuve élémentaires.

Définition 3.1.1 *Soit $\mathcal{S} = (A, F, R)$ un système formel. Un* **arbre de preuve élémentaire** *est un arbre de la forme $\dfrac{\pi_1 \quad \dots \quad \pi_n}{\varphi}\ \iota$, noté $(\pi_1, \dots, \pi_n, \varphi)_\iota$, tel que, pour tout $i, 1 \leq i \leq n$:*

- *soit π_i est une formule de F,*
- *soit il existe une règle $\iota_i \in R$ telle que $\pi_i = (\varphi'_{i1}, \dots, \varphi'_{im_i}, \varphi_i)_{\iota_i}$ avec pour tout $j, 1 \leq j \leq m_i$, φ'_{im_j} une formule de F.*

Remarque 3.1.1 *Les arbres de preuve élémentaires ne sont pas des arbres de preuve car leurs feuilles ne sont pas nécessairement des axiomes.*

3.1.3 Procédure de transformation d'arbres de preuve

A partir de ces arbres de preuve élémentaires nous définissons nos transformations d'arbres de preuve. Une transformation doit être vue comme une règle de réécriture d'arbre qui a pour membre gauche un arbre de preuve élémentaire π qui correspond à ce que l'on veut transformer dans un arbre, et pour membre droit un arbre de preuve π' de même conclusion que π et dont les feuilles sont un sous-ensemble de celle de π. La partie droite doit bien sûr être un arbre normalisé.

Définition 3.1.2 (Système de transformation) *Soit $\mathcal{S} = (A, F, R)$ un système formel. Un* **système de transformation d'arbre de preuve** *pour $\mathcal{S}$ est une relation binaire $\rightsquigarrow$ telle que pour toute* **règle de transformation** *de la forme $\pi \rightsquigarrow \pi'$:*

- *π est un arbre de preuve élémentaire,*
- *π' est un arbre normalisé, c'est-à-dire qu'on ne peut pas appliquer de règles sur celui-ci,*

- *l'ensemble des feuilles de π' est inclus dans celui de π (noté $\mathcal{LS}(\pi') \subseteq \mathcal{LS}(\pi)$),*
- *π et π' ont la même conclusion.*

Nous pouvons transformer tout système de transformation d'arbre de preuve en un système de réécriture. Ceci nous permettra de montrer la normalisation d'arbres de preuve à partir des nombreux résultats établis sur la terminaison des systèmes de réécriture.

Nous allons donc nous ramener à un système de réécriture dont nous avons étudié de nombreuses propriétés dans le chapitre précédent. Pour cela, nous allons associer à toute formule φ une sorte s_φ, et à chaque règle d'inférence ι une opération f_ι. Pour les arbres qui ne sont pas des arbres de preuve, et qui possèdent donc des feuilles qui ne sont pas des axiomes, nous allons remplacer ces feuilles par des variables. Associons au système formel $\mathcal{S} = (A, F, R)$ la signature $\Sigma_\mathcal{S} = (S_\mathcal{S}, F_\mathcal{S}, V_\mathcal{S})$ telle que :

- $S_\mathcal{S} = \{s_\varphi \mid \varphi \in F\}$
- $F_\mathcal{S} = \{c_\varphi :\rightarrow s_\varphi \mid \varphi \in F\} \cup \{f_\iota : s_{\varphi_1} \times \cdots \times s_{\varphi_n} \rightarrow s_\varphi \mid \frac{\varphi_1 \ldots \varphi_n}{\varphi}\iota \in R\}$
- $V_\mathcal{S} = \{x_\varphi \mid \varphi \in F\}$.

Nous pouvons donc associer à tout arbre de preuve de $\mathcal{S}$ un $\Sigma_\mathcal{S}$-terme clos. Ainsi, un système de transformations élémentaires d'arbres de preuve peut être vu comme un système de réécriture. À toute règle de transformation d'arbre de preuve

$$\dfrac{\psi_1 \quad \ldots \quad \dfrac{\varphi_1 \quad \ldots \quad \varphi_n}{\varphi'}\,\iota' \quad \ldots \quad \psi_m}{\varphi}\,\iota \quad \rightsquigarrow \quad \pi,$$

nous associons une règle de réécriture de termes :

$$f_\iota(x_{\psi_1}, \ldots, f_{\iota'}(x_{\varphi_1}, \ldots, x_{\varphi_n}), \ldots, x_{\psi_m}) \rightsquigarrow t_\pi$$

telle que $x_{\psi_1}, \ldots, x_{\psi_m}, x_{\phi_1}, \ldots, x_{\phi_m}$ sont des variables de sortes s_{ψ_j} (resp. s_{ϕ_j}), et t_π est le $\Sigma_\mathcal{S}$-terme obtenu à partir de π où toute feuille de φ a été remplacée dans t_π par une variable x_φ de sorte s_φ.

Définition 3.1.3 (Étape de transformation) *Soit $\mathcal{T}$ un système de transformation d'arbre de preuve. Soit $\mathcal{TR}$ le système de réécriture de terme associé à $\mathcal{T}$.*

$\rightsquigarrow_{\mathcal{TR}}$ est la relation associée à l'application d'une étape de réécriture du système $\mathcal{TR}$. Comme nous avons traduit nos arbres de preuve en termes, une **étape de transformation** *se fera de la même manière qu'une étape de réécriture (voir définition 2.3.2 page 48).*

On notera $\overset{*}{\rightsquigarrow}_{\mathcal{TR}}$ l'application successive de plusieurs étape transformation d'arbre de preuve.

Comme pour tous les systèmes de réécriture l'un des problèmes majeurs est la terminaison.

Définition 3.1.4 *Un système de réécriture est dit* **fortement normalisant** *si toutes les séquences de transformation sont finies, c'est-à-dire qu'il termine quelle que soit la manière d'appliquer les règles, et* **faiblement normalisant** *si tout arbre de preuve a une forme normale.*

Nous ne nous occuperons pas ici de la confluence du système de réécriture. Seule la forme de l'arbre normalisé nous intéresse, peu importe que celle-ci soit unique ou non.

3.2 Conditions et théorème pour la normalisation forte

Cette section est dédiée à un théorème de normalisation forte sur les transformations d'arbres de preuve, c'est-à-dire un théorème qui garantit la terminaison d'un système de transformation d'arbres de preuve sous certaines conditions (que nous allons décrire dans cette section). Ce théorème est général parce qu'il est défini indépendamment d'un système formel donné (voir section 1.2.2).

Une idée simple et commune à la plupart des preuves de terminaison (voir l'ordre récursif sur les chemins section 2.4.2) consiste à comparer les termes des règles de réécriture en commençant par comparer leurs symboles de tête, et puis récursivement en comparant les collections de leurs sous termes immédiats. Nous commençons donc par définir un ordre bien-fondé sur les éléments atomiques des arbres de preuve (i.e. les instances de règles).

Par exemple, dans le cadre du calcul des séquents, et comme nous le verrons par la suite dans la section 3.3.3 page 82 pour prouver l'élimination des coupures, nous pouvons définir l'ordre bien fondé suivant sur les règles du calcul :

$$\forall @ \in \{\wedge, \neg, \exists\}, Coupure \succ @Droite, @Gauche, Axiome$$

En classant les règles à l'aide de cet ordre bien-fondé, nous pouvons caractériser une forme d'arbres de preuve du calcul des séquents, où la règle de coupure "remonte" sur toutes les règles et s'élimine sous les axiomes. Lorsqu'on écrit $Cut \succ \wedge Gauche$ cela signifie que la règle Cut peut remonter sur la règle $\wedge Gauche$, et ceci nous permet d'accepter cette règle de transformation [1] :

[1] L'ordre bien-fondé $\succ$ ne nous garantit pas que cette règle existe mais qu'elle peut exister.

$$\dfrac{\Gamma_3 \rightarrowtail \Delta_2, \varphi \quad \dfrac{A, B, \varphi \rightarrowtail \Delta_1}{A \wedge B, \varphi \rightarrowtail \Delta_1}\wedge Gauche}{\Gamma_3, A \wedge B \rightarrowtail \Delta_1', \Delta_2} Cut \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma_3 \rightarrowtail \Delta_2, \varphi \quad A, B, \varphi \rightarrowtail \Delta_1}{\Gamma_3, A, B \rightarrowtail \Delta_1, \Delta_2} Cut}{\Gamma_3, A \wedge B \rightarrowtail \Delta_1, \Delta_2}\wedge Gauche$$

Nous allons maintenant donner plus précisément nos conditions pour qu'une procédure de transformation soit fortement normalisante. Nous allons représenter les arbres de preuve sous la forme de déductions (voir section 1.2.2).

Définition 3.2.1 (Conditions pour la normalisation forte) *Soit $\mathcal{S} = (A, F, R)$ un système formel. Soit $\mathcal{T}$ un système de transformation, et $\preceq$ un ordre bien-fondé sur l'ensemble des règles d'inférences R.*

Un procédure de transformation $\rightsquigarrow_{\mathcal{T}}$ vérifie les **conditions pour la normalisation forte** *si pour toute règle de transformation $(\iota_1, \ldots, \iota_n, \varphi)_\iota \rightsquigarrow \pi \in \mathcal{T}$ et pour chaque sous-arbre $(\iota'_1, \ldots, \iota'_m, \varphi')_{\iota'}$ de π :*

1. $\iota \succeq \iota'$ *si pour tout $j, 1 \leq j \leq m$, on a $\iota'_j \in F \cup \bigcup_{r \in R} r$*

   $\iota \succ \iota'$ *sinon*
2. *Si $\iota \sim \iota'$, alors $\{\{\iota_1, \ldots, \iota_n\}\} \gg \{\{\iota'_1, \ldots, \iota'_m\}\}$ où $\gg$ est l'extension de $\succ$ aux multi-ensembles[2] de $F \cup \bigcup_{r \in R} r$.*

Notons que ces conditions sont très simples à vérifier sur un système de transformations d'arbres de preuve.

Théorème 3.2.1 *Toute procédure de transformation $\rightsquigarrow_{\mathcal{T}}$ satisfaisant les conditions de la définition 3.2.1 est fortement normalisante.*

Preuve Les arbres de preuve sont vus comme des termes en utilisant la transformation définie dans la section précédente. Soient $\rightsquigarrow$ un système de règles de réécriture qui vérifie les conditions pour la normalisation forte, et $>^{rpo}$ un ordre récursif sur les chemins (Recursive Path Ordering en anglais) muni d'une relation d'ordre bien-fondé $\succeq$ sur l'ensemble de règles.

Montrons que $\rightsquigarrow \subseteq >^{rpo}$.

Soit $(\iota_1, \ldots, \iota_n, \varphi)_\iota \rightsquigarrow \pi$ une règle de transformation. Montrons par induction sur π que $(\iota_1, \ldots, \iota_n, \varphi)_\iota >^{rpo} \pi$.

Cas de base π est une formule φ.

D'après la définition 3.1.1, $\mathcal{LS}(\pi) \subseteq \mathcal{LS}((\iota_1, \ldots, \iota_n, \varphi)_\iota)$, ce qui signifie que π est un sous-terme de $(\iota_1, \ldots, \iota_n, \varphi)_\iota$. Comme l'ordre récursif sur les chemins a la propriété de sous-termes alors $(\iota_1, \ldots, \iota_n, \varphi)_\iota >^{rpo} \pi$.

[2] Voir section 2.4.1.3 page 53 pour une description détaillée des multi-ensembles.

Étape d'induction π est de la forme $(\pi_1, \ldots, \pi_n, \varphi)_{\iota'}$.

Deux cas de figure sont possibles :

1. $\iota \succ \iota'$. Par hypothèse d'induction, on sait que pour tout i, $1 \leq i \leq n$, $(\iota_1, \ldots, \iota_n, \varphi)_\iota >^{rpo} \pi_i$, donc par le RPO, $(\iota_1, \ldots, \iota_n, \varphi)_\iota >^{rpo} \pi$, car $\iota \succ \iota'$.
2. $\iota \sim \iota'$. Par la condition 1, chaque π_i est dans $F \cup \bigcup_{r \in R} r$ ($1 \leq i \leq n$). Par la condition 2, nous avons $\{\{\iota_1, \ldots, \iota_n\}\} \not\succ \{\{\iota'_1, \ldots, \iota'_m\}\}$, et donc $\{\{\iota_1, \ldots, \iota_n\}\} \gg^{rpo} \{\{\iota'_1, \ldots, \iota'_m\}\}$. Par le RPO, nous pouvons dire que $(\iota_1, \ldots, \iota_n, \varphi)_\iota >^{rpo} \pi$.

★

3.3 Exemples de normalisation forte

Dans cette section, nous allons appliquer notre résultat de normalisation forte, dont le point central est le théorème 3.2.1, à trois résultats bien connus. Ces résultats ont pour caractéristique d'avoir des preuves non triviales, mais qui peuvent être vues comme corollaires simples de ce théorème. Ce sont des instances particulières du résultat général de terminaison énoncé par ce théorème.

Le premier de ces résultats est lié à la complétude du raisonnement algébrique par rapport au raisonnement équationnel, connu sous le nom de logicalité. Le deuxième est un résultat bien connu en réécriture : le lemme de Newman. Le troisième résultat de normalisation auquel nous allons nous intéresser est attaché au calcul des séquents de Gentzen pour lequel les règles structurelles sont implicites (présentation dans la section 1.2.3.1) et au fait que l'on peut éliminer la règle de coupure dans tous les arbres de preuve.

3.3.1 Logicalité

Dans le cadre de la logique équationnelle (voir section 1.3 page 32) on dispose d'un résultat connu sous le nom de logicalité [Bir35, Bah03]. Nous allons le définir en guise d'exemple d'application pour notre cadre. Ce résultat est lié à la définition d'une relation de convertibilité. Pour commencer nous devons définir le système formel équationnelle sur lequel nous allons travailler.

3.3.1.1 Système formel équationnel

Lorsque les formules ne sont que des équations de la forme $t = t'$ avec $t, t' \in T_\Sigma(V)$ nous pouvons définir le système formel équationnel $\mathcal{S}_{\mathcal{LE}}$ associé de la façon

suivante :

Définition 3.3.1 *Le système formel équationnel $\mathcal{S}_{\mathcal{LE}} = (A_{\mathcal{LE}}, F_{\mathcal{LE}}, R_{\mathcal{LE}})$ associé à une signature $\Sigma = (S, F, V)$ est défini par :*

- $A_{\mathcal{LE}} = V \cup F \cup \{=\}$
- $F_{\mathcal{LE}} = Eq(\Sigma)$
- *l'ensemble $R_{\mathcal{LE}}$ contient les cinq schémas de règles d'inférence suivants, instanciables en remplaçant s, t, t' et t'' par des termes quelconques de $T_\Sigma(V)$ et σ par toute substitution de V dans $T_\Sigma(V)$:*

$$refl\ \frac{}{t = t} \qquad sym\ \frac{t = t'}{t' = t}$$

$$trans\ \frac{t = t' \quad t' = t''}{t = t''}$$

$$rempl\ \frac{t = t'}{s[t]_w = s[t']_w}$$

$$subst\ \frac{t = t'}{\sigma(t) = \sigma(t')}$$

3.3.1.2 Relation de convertibilité

À partir du calcul $\mathcal{LE}$, on observe très vite que le raisonnement défini au travers de ce calcul ne caractérise pas le raisonnement algébrique usuel. La conséquence de l'utilisation de ce calcul est que les preuves obtenues, même pour prouver des choses simples, sont souvent lourdes et ne correspondent pas à la démarche habituelle comme le montre cet exemple :

Exemple 3.3.1 *Soit Σ une signature composée de la seule opération binaire $\_+\_$, et Ax un ensemble de formules composé des deux axiomes $x+(y+z) = (x+y)+z$ qui caractérise l'associativité, et $x + y = y + x$ la commutativité de l'opération $+$.*

Montrons alors que la formule $(a + b) + (a + b) = (a + a) + (b + b)$ avec $a, b \in V$ est un théorème de $Th_{\mathcal{LE}}(Ax)$:

$$\dfrac{\dfrac{x + (y + z) = (x + y) + z}{(a + b) + (a + b) = ((a + b) + a) + b}\ subst \qquad \mathcal{A}_1}{(a + b) + (a + b) = (a + a) + (b + b)}\ trans$$

$\mathcal{A}_1$ étant le sous-arbre de preuve suivant :

$$\dfrac{\mathcal{A}_2 \qquad \dfrac{x+(y+z)=(x+y)+z}{((a+a)+b)+b=(a+a)+(b+b)}\ subst}{((a+b)+a)+b=(a+a)+(b+b)}\ trans$$

$\mathcal{A}_2$ étant le sous-arbre de preuve suivant :

$$\dfrac{\dfrac{x+y=y+x}{(a+b)+a=a+(a+b)}\ subst \qquad \dfrac{x+(y+z)=(x+y)+z}{a+(a+b)=((a+a)+b)}\ subst}{\dfrac{((a+b)+a)=((a+a)+b)}{((a+b)+a)+b=((a+a)+b)+b}\ rempl}\ trans$$

Toutes les feuilles de l'arbre sont des axiomes de l'ensemble Ax, la formule $(a+b)+(a+b)=(a+a)+(b+b)$ est donc bien un théorème.

Pour mieux appréhender le raisonnement algébrique usuellement employé, on définit alors une relation binaire $\leftrightarrow_\Gamma$ sur $T_\Sigma(V)$, appelée **relation de convertibilité**, de la façon suivante :

Définition 3.3.2 (Convertibilité) *Soient $\Sigma = (F, ar)$ une signature et V un ensemble de variables. Soit Γ un ensemble d'équations. Notons $\leftrightarrow_\Gamma$ la relation binaire sur $T_\Sigma(V)$ définie comme étant la plus petite relation (au sens de l'inclusion) telle que :*

1. *$\Gamma \subseteq \leftrightarrow_\Gamma$, c'est-à-dire que si $u = v \in \Gamma$ alors $u \leftrightarrow_\Gamma v$*
2. *si $t \leftrightarrow_\Gamma t'$ alors $t' \leftrightarrow_\Gamma t$,*
3. *si $t \leftrightarrow_\Gamma t'$ alors, pour toute substitution $\sigma : V \to T_\Sigma(V)$, $\sigma(t) \leftrightarrow_\Gamma \sigma(t')$ et*
4. *si $t \leftrightarrow_\Gamma t'$ alors, pour tout contexte $c[.]_w$ de $T_\Sigma(V)$, $c[t]_w \leftrightarrow_\Gamma c[t']_w$.*

La relation définie ci-dessus représente une étape de raisonnement algébrique. La composition de plusieurs de ces étapes se définit naturellement par la fermeture réflexive et transitive de la relation $\leftrightarrow_\Gamma$, c'est-à-dire $\overset{*}{\leftrightarrow}_\Gamma$.
*$\overset{*}{\leftrightarrow}_\Gamma$ est appelée **relation de convertibilité**.*

Exemple 3.3.2 *Pour prouver la formule $(a+b)+(a+b) = (a+a)+(b+b)$ de l'exemple 3.3.1 page 75) à l'aide de la relation de convertibilité, il suffit d'appliquer la stratégie suivante :*
$(a+b)+(a+b) \leftrightarrow_{Ax} ((a+b)+a)+b \leftrightarrow_{Ax} (a+(a+b))+b \leftrightarrow_{Ax} ((a+a)+b)+b \leftrightarrow_{Ax} (a+a)+(b+b)$

Contrairement à la construction d'un arbre de preuve, il faut très peu d'étapes pour obtenir une preuve de $(a+b)+(a+b) \overset{*}{\leftrightarrow}_{Ax} (a+a)+(b+b)$.

3.3.1.3 Résultat de logicalité

Nous pouvons constater que cette définition de la relation de convertibilité $\overset{*}{\leftrightarrow}_\Gamma$ désigne des arbres de preuves d'une forme particulière. En effet, cette relation est construite à partir de l'ensemble Γ et à l'aide des règles d'inférence de la logique équationnelle :

- le point 1. de la définition 3.3.2 correspond aux axiomes,
- le point 2. correspond la règle sym,
- le point 3. correspond la règle $subst$,
- le point 4. correspond à une composition de la règle $rempl$,
- et la fermeture réflexive et transitive correspond une composition des règles $refl$ et $trans$.

Pour chaque expression $t \overset{*}{\leftrightarrow}_\Gamma t'$, il existe donc un arbre de preuve (voir figure 2.1) dont les feuilles appartiennent à Γ et qui est :

- ou bien réduit à une instance de $refl$,
- ou bien composé uniquement d'instances de $sym,\ subst,\ rempl$ et $trans$ et tel qu'une instance de $trans$ ne se trouve jamais au-dessus d'une instance de $rempl$, de $subst$ et de sym.

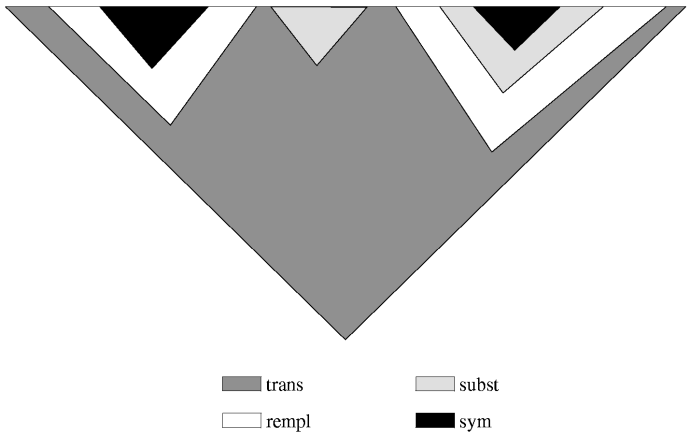


FIG. 3.1 – Arbre de preuve associé à $\overset{*}{\leftrightarrow}_\Gamma$

La définition de la règle de convertibilité traduit alors une stratégie d'application des règles d'inférence. Nous devons alors vérifier que cette stratégie ne perd pas en puissance de preuve, c'est-à-dire qu'elle est correcte et complète. C'est le résultat de logicalité que l'on peut exprimer de la façon suivante :

Théorème 3.3.1 (Logicalité) *Étant donné un ensemble d'équations* Γ*, nous avons l'équivalence suivante :*

$$t \overset{*}{\leftrightarrow}_{\Gamma} t' \textit{ si et seulement si } \Gamma \vdash t = t'$$

Le sens ($\Rightarrow$) de l'équivalence est trivialement vérifié car la relation de convertibilité traduit un arbre de preuve particulier explicité ci-dessus.

L'autre sens ($\Leftarrow$) de l'équivalence est plus complexe car il demande qu'à tout énoncé de $\Gamma \vdash t = t'$, il existe un arbre de preuve satisfaisant la stratégie définie précédemment. À l'aide de notre cadre général, nous pouvons transformer n'importe quel arbre de preuve en un arbre possédant la structure recherchée.

3.3.1.4 Modélisation dans notre cadre général

Soit $\mathcal{S}_{\Sigma,V} = (\mathcal{A}_{\Sigma,V}, \mathcal{F}_{\Sigma,V}, \mathcal{R}_{\Sigma,V})$ le système formel équationnel de la définition 3.3.1 page 75. Nous voulons construire des arbres dans lesquels les instances de $trans$ ne se trouvent jamais au-dessus des celles de sym, $subst$, et $rempl$. Pour cela, nous définissons un système de transformation d'arbres de preuve via les règles de normalisation suivantes :

1. **subst** remonte sur **trans** :

$$\dfrac{\dfrac{t = t' \qquad t' = t''}{t = t''}\, trans}{\sigma t = \sigma t''}\, subst \quad \rightsquigarrow \quad \dfrac{\dfrac{t = t'}{\sigma t = \sigma t'}\, subst \qquad \dfrac{t' = t''}{\sigma t' = \sigma t''}\, subst}{\sigma t = \sigma t''}\, trans$$

2. **rempl** remonte sur **trans** :

$$\dfrac{\dfrac{t = t' \qquad t' = t''}{t = t''}\, trans}{s[t]_w = s[t'']_w}\, rempl \quad \rightsquigarrow \quad \dfrac{\dfrac{t = t'}{s[t]_w = s[t']_w}\, rempl \qquad \dfrac{t' = t''}{s[t']_w = s[t'']_w}\, rempl}{s[t]_w = s[t'']_w}\, trans$$

3. **sym** remonte sur **trans** :

$$\dfrac{\dfrac{t = t' \qquad t' = t''}{t = t''}\, trans}{t'' = t}\, sym \quad \rightsquigarrow \quad \dfrac{\dfrac{t' = t''}{t'' = t'}\, sym \qquad \dfrac{t = t'}{t' = t}\, sym}{t'' = t}\, trans$$

On remarque également que les règles $subst$, $rempl$ et sym s'annulent toutes sous $refl$ et $refl$ et $trans$ s'annulent mutuellement. C'est ce que l'on montre avec les règles de transformation suivantes :

$$subst\ \frac{refl\ \frac{}{t=t}}{\sigma(t)=\sigma(t)} \rightsquigarrow refl\ \frac{}{\sigma(t)=\sigma(t)}$$

$$rempl\ \frac{refl\ \frac{}{t=t}}{s[t]_w=s[t]_w} \rightsquigarrow refl\ \frac{}{s[t]_w=s[t]_w}$$

$$sym\ \frac{refl\ \frac{}{t=t}}{t=t} \rightsquigarrow refl\ \frac{}{t=t}$$

$$trans\ \frac{refl\ \frac{}{t=t} \quad \pi : t=t'}{t=t'} \rightsquigarrow \pi : t=t' \qquad trans\ \frac{\pi : t=t' \quad refl\ \frac{}{t'=t'}}{t=t'} \rightsquigarrow \pi : t=t'$$

Comme nous l'avons déjà observé, les règles de transformation consistent à "distribuer" les instances des règles $sym, subst$ et $rempl$ sur les instances de $trans$, et à éliminer toutes les instances de règles quand elles se trouvent en dessous de la règle $refl$. Nous considérons alors un ordre bien fondé sur l'ensemble des règles du système formel tel que $sym \sim subst \sim rempl \succ trans \succ refl \succ Axiom$. Il est facile de vérifier que les règles de transformation respectent les conditions de normalisation fortes de la définition 3.2.1 page 73. D'après le théorème 3.2.1, ceci assure alors la terminaison de la relation $\rightsquigarrow$.

3.3.2 Lemme de Newman

Dans cette section nous allons donner une preuve originale du lemme de Newman énoncé dans la définition 2.3.1 page 50 en utilisant l'approche que nous avons développé dans ce chapitre.

3.3.2.1 La réécriture (système formel)

Pour pouvoir s'intégrer dans notre cadre de normalisation il faut tout d'abord présenter la réécriture de termes sous la forme d'un système formel.

Définition 3.3.3 *Soit $\mathcal{R}$ un système de réécriture. Le système formel associé*[3] *$\mathcal{S}_\mathcal{R} = (F, R)$ est défini de la manière suivante :*

[3]Ne pas confondre $\mathcal{R}$ le système de réécriture et R l'ensemble des règles d'inférence.

– *F est l'ensemble de toutes les formules de la forme $u \to_{\mathcal{R}} v$, $u \overset{*}{\to}_{\mathcal{R}} v$ et $u \overset{*}{\leftrightarrow}_{\mathcal{R}} v$,*
– *R est l'ensemble des règles d'inférences suivant :*

$$\frac{u \to v \in \mathcal{R}}{u \to_{\mathcal{R}} v}\ Axiom \qquad \frac{u \to_{\mathcal{R}} v}{u \overset{*}{\to}_{\mathcal{R}} v}\ Rew. \qquad \frac{u \overset{*}{\to}_{\mathcal{R}} v}{u \overset{*}{\leftrightarrow}_{\mathcal{R}} v}\ Der/Pr.$$

$$\frac{u \overset{*}{\to}_{\mathcal{R}} v}{C[u] \overset{*}{\to}_{\mathcal{R}} C[v]}\ Cont. \qquad \frac{u \overset{*}{\to}_{\mathcal{R}} v}{\sigma(u) \overset{*}{\to}_{\mathcal{R}} \sigma(v)}\ Sub.$$

$$\frac{u \ \mathcal{R} \overset{*}{\leftarrow} w \qquad w \overset{*}{\to}_{\mathcal{R}} v}{u \overset{*}{\leftrightarrow}_{\mathcal{R}} v}\ Peak \qquad \frac{u \overset{*}{\to}_{\mathcal{R}} w \qquad w \ \mathcal{R} \overset{*}{\leftarrow} v}{u \overset{*}{\leftrightarrow}_{\mathcal{R}} v}\ Valley$$

$$\frac{u \overset{*}{\leftrightarrow}_{\mathcal{R}} w \qquad w \overset{*}{\leftrightarrow}_{\mathcal{R}} v}{u \overset{*}{\leftrightarrow}_{\mathcal{R}} v}\ Proof \qquad \frac{u \overset{*}{\to}_{\mathcal{R}} w \qquad w \overset{*}{\to}_{\mathcal{R}} v}{u \overset{*}{\to}_{\mathcal{R}} v}\ Deriv.$$

Exemple 3.3.3 *Reprenons l'exemple simple, déjà étudié, du système de réécriture $\mathcal{R}_{fig}$ composé des deux règles suivantes :*

$carre(triangle(f)) \to triangle(carre(f))$
$cercle(triangle(f)) \to triangle(f)$

Grâce à ce système de réécriture, nous pouvons écrire :

$$cercle(carre(triangle(rien))) \overset{*}{\to}_{\mathcal{R}_{fig}} triangle(carre(rien))$$

Grâce au système formel $\mathcal{S}_{\mathcal{R}}$ nous pouvons construire l'arbre de preuve correspond à cette expression[4] *:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{ca(t(u)) \overset{*}{\to}_{\mathcal{R}_{fig}} t(ca(u))}\ Axiom}{ce(ca(t(u))) \overset{*}{\to}_{\mathcal{R}_{fig}} ce(t(ca(u)))}\ Cont. \qquad \dfrac{\dfrac{}{ce(t(v)) \overset{*}{\to}_{\mathcal{R}_{fig}} t(v)}\ Axiom}{ce(t(ca(u))) \overset{*}{\to}_{\mathcal{R}_{fig}} t(ca(u))}\ Sub.}{ce(ca(t(u))) \overset{*}{\to}_{\mathcal{R}_{fig}} t(ca(u))}\ Deriv}{ce(ca(t(rien))) \overset{*}{\to}_{\mathcal{R}_{fig}} t(ca(rien))}\ Sub.$$

Ce système de preuve va nous permettre de décrire une procédure de transformation qui termine et qui transforme une preuve qui contient des pics ($Peak$) en une preuve qui n'en contient plus aucun. Les pics locaux vont être remplacés étape par étape par des vallées équivalentes.

[4] Pour alléger les notations, on notera *cercle* en *ce*, *carre* en *ca*, et *triangle* en *t*.

Voici les règles de transformation élémentaires :

$$Peak\frac{Deriv.\frac{u\ \mathcal{R}^{*}\!\leftarrow u'\ \mathcal{R}^{*}\!\leftarrow w}{u\ \mathcal{R}^{*}\!\leftarrow w}\ w\ \overset{*}{\rightarrow}_{\mathcal{R}}\ v}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v} \rightsquigarrow Proof\frac{Der/Pr\frac{u\ \mathcal{R}^{*}\!\leftarrow u'}{u\overset{*}{\leftrightarrow}_{\mathcal{R}}u'}\ Peak\frac{u'\ \mathcal{R}^{*}\!\leftarrow w\overset{*}{\rightarrow}_{\mathcal{R}}v}{u'\overset{*}{\leftrightarrow}_{\mathcal{R}}v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v}$$

$$Peak\frac{u\ \mathcal{R}^{*}\!\leftarrow w\ Deriv.\frac{w\overset{*}{\rightarrow}_{\mathcal{R}}v'\ \mathcal{R}^{*}\!\leftarrow v}{w\overset{*}{\rightarrow}_{\mathcal{R}}v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v} \rightsquigarrow Proof\frac{Peak\frac{u\ \mathcal{R}^{*}\!\leftarrow w\overset{*}{\rightarrow}_{\mathcal{R}}v'}{u\overset{*}{\leftrightarrow}_{\mathcal{R}}v'}\ Der/Pr\frac{v'\ \mathcal{R}^{*}\!\leftarrow v}{v'\overset{*}{\leftrightarrow}_{\mathcal{R}}v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v}$$

$$Peak\frac{Rew.\frac{u\ \mathcal{R}\leftarrow w\rightarrow_{\mathcal{R}}v}{u\ \mathcal{R}^{*}\!\leftarrow w\ \mathcal{R}^{*}\!\rightarrow v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v} \rightsquigarrow Valley\frac{u\ \overset{*}{\rightarrow}_{\mathcal{R}}\ x\ \mathcal{R}^{*}\!\leftarrow v}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v}$$

$$Proof\frac{Valley\frac{u\overset{*}{\rightarrow}_{\mathcal{R}}x\ \mathcal{R}^{*}\!\leftarrow w}{u\overset{*}{\leftrightarrow}_{\mathcal{R}}w}\ Der/Pr\frac{w\overset{*}{\rightarrow}_{\mathcal{R}}v}{w\overset{*}{\leftrightarrow}_{\mathcal{R}}v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v} \rightsquigarrow Proof\frac{Der/Pr\frac{u\overset{*}{\rightarrow}_{\mathcal{R}}x}{u\overset{*}{\leftrightarrow}_{\mathcal{R}}x}\ Peak\frac{x\ \mathcal{R}^{*}\!\leftarrow w\overset{*}{\rightarrow}_{\mathcal{R}}v}{x\overset{*}{\leftrightarrow}_{\mathcal{R}}v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v}$$

$$Proof\frac{Der/Pr\frac{u\ \mathcal{R}^{*}\!\leftarrow u'}{u\overset{*}{\leftrightarrow}_{\mathcal{R}}u'}\ Valley\frac{u'\overset{*}{\rightarrow}_{\mathcal{R}}x\ \mathcal{R}^{*}\!\leftarrow v}{u'\overset{*}{\leftrightarrow}_{\mathcal{R}}v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v} \rightsquigarrow Proof\frac{Peak\frac{u\ \mathcal{R}^{*}\!\leftarrow u'\overset{*}{\rightarrow}_{\mathcal{R}}x}{u\overset{*}{\leftrightarrow}_{\mathcal{R}}x}\ Der/Pr\frac{x\overset{*}{\leftarrow}_{\mathcal{R}}v}{x\overset{*}{\leftrightarrow}_{\mathcal{R}}v}}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v}$$

Cet ensemble de règles de transformation décrit une classe particulière d'arbre de preuve où toutes les instances des règles Peak et Proof remontent sur toutes les autres règles du système formel.

La procédure de transformation d'arbres de preuve $\rightsquigarrow$ définie par les règles de transformation que nous venons de décrire satisfait les conditions pour la normalisation pour l'ordre bien-fondé $\preceq$ définit par :

$$Peak, Proof \succ Rew \sim Cont. \sim Der/Pr. \sim Sub. \sim Valley \sim Deriv.$$

$$\iota\frac{u_1\ @\ u_2\ @\ u_3}{u_1\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ u_3} \succ \iota'\frac{u'_1\ @'\ u'_2\ @'\ u'_3}{u'_1\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ u'_3} \Longleftrightarrow u_2\ \overset{+}{\rightarrow}_{\mathcal{R}}\ u'_2 \wedge \forall i = 1,2,3\ \ u_i\ \overset{*}{\rightarrow}_{\mathcal{R}}\ u'_i$$

$\iota, \iota' \in Proof \cup Peak$ et $@, @' \in \{\overset{*}{\leftrightarrow}_{\mathcal{R}}, \overset{*}{\rightarrow}_{\mathcal{R}}\}$.

$$Peak\frac{u_1\ \mathcal{R}\overset{*}{\leftarrow} u_2\ \overset{*}{\rightarrow}_{\mathcal{R}}\ u_3}{u\ \overset{*}{\leftrightarrow}_{\mathcal{R}}\ v} \succ u\ \overset{*}{\rightarrow}_{\mathcal{R}}\ v \Longleftrightarrow u_2\ \overset{+}{\rightarrow}_{\mathcal{R}}\ u \wedge \exists j = 1,3 u_j\ \overset{*}{\rightarrow}_{\mathcal{R}}\ u$$

3.3.3 Élimination des coupures

L'élimination des coupures dans le calcul des séquents est un résultat bien connu et sûrement un des plus important en théorie de la démonstration [Gal86, Kle52]. Il nous dit que s'il existe une preuve d'une formule quelconque dans le calcul des séquents alors il existe aussi une preuve de cette formule dans le même calcul sans la règle de Coupure. Dans [Gen35], Gentzen a montré que toutes les coupures peuvent être éliminées dans des arbres de preuve pour une version classique du calcul des séquent où les règles structurelles sont explicites. Il a donné une preuve constructive du résultat d'élimination des coupures, en définissant un procédure pour éliminer les coupures dans une preuve π dont la conclusion est une tautologie A et dont les feuilles sont des axiomes. Il obtient ainsi une preuve π' qui prouve A à partir d'axiomes sans application de la règle de coupure. Cette procédure consiste à transformer l'arbre à l'aide de transformations élémentaire d'arbres de preuve. Nous allons donc pouvoir la modéliser facilement dans notre cadre abstrait.

3.3.3.1 Le calcul des séquents LK

Nous allons considérer notre version du calcul des séquents vu dans la section 1.2.3 pour la logique des prédicats du premier ordre. Pour limiter le nombre de transformations élémentaires dans la procédure de normalisation, nous allons considérer un sous-ensemble des formules du premier ordre en nous restreignant aux connecteurs logique $\{\neg, \vee\}$ et au quantificateur existentielle $\exists$. Tous les autres connecteurs peuvent être redéfinis à partir de ce sous-ensemble.

Définition 3.3.4 *Soient A une formule quelconque de la logique des prédicats du premier ordre. La* **formule simplifiée** $[A]$ *de la formule quelconque A est construite de la manière suivante :*

- *si A est une formule sans connecteurs, alors $[A] = A$;*
- *si $A = \neg B$, avec B une formule quelconques, alors $[A] = \neg[B]$;*
- *si $A = B \vee C$, avec B et C deux formules quelconques, alors $[A] = [B] \vee [C]$;*
- *si $A = \exists x.B$, avec B une formule quelconque et x une variable, alors $[A] = \exists x.[B]$;*
- *si $A = B \Rightarrow C$, avec B et C deux formules quelconques, alors $[A] = (\neg[B]) \vee [C]$;*
- *si $A = (B \wedge C)$, avec B et C deux formules quelconques, alors $[A] = \neg((\neg B) \vee (\neg C))$;*
- *si $A = \forall x.B$, avec B une formule quelconque et x une variable, alors $[A] = \neg\exists x.(\neg[B])$.*

Les séquents de ce calcul vont être construits à partir de formules simplifiées. Comme pour la logique des prédicats du premier ordre, nous considérons des ensembles plutôt que des multi-ensembles ou des listes de formules pour construire les séquents. C'est un avantage puisque la contraction (inutile en utilisant les ensembles) pose des problème pour l'élimination des coupures.

Définition 3.3.5 *Soient Γ et Δ deux ensembles de formules simplifiées.*
Un **séquent** *est une formule $\Gamma \rightarrowtail \Delta$.*

Le calcul est simplement une restriction de celui présenté définition 1.2.11 page 30. On le représente à l'aide d'un système formel.

Définition 3.3.6 *Soient $\Sigma = (S, F, R, V)$ une signature du premier ordre. $S_{LK} = (A_{LK}, F_{LK}, R_{LK})$ est notre système formel pour les séquents simplifiées tel que*

- *A_{LK} est l'alphabet $F \cup R \cup V \cup \{\neg, \wedge, \vee, \Rightarrow, \forall, \exists\} \cup \{\rightarrowtail\}$,*
- *F_{LK} contient tous les séquents bien formés que l'on peut construire à partir de A_{LK},*
- *R_{LK} est l'ensemble des instances des schémas de règles suivant :*

$$\frac{}{\Gamma, \varphi \rightarrowtail \Delta, \varphi}\textit{Ax.}$$

$$\frac{\Gamma, \varphi \rightarrowtail \Delta \quad \Gamma, \varphi' \rightarrowtail \Delta}{\Gamma, \varphi \vee \varphi' \rightarrowtail \Delta}\vee\textit{Gauche} \qquad \frac{\Gamma \rightarrowtail \Delta, \varphi, \varphi'}{\Gamma \rightarrowtail \Delta, \varphi \vee \varphi'}\vee\textit{Droite}$$

$$\frac{\Gamma \rightarrowtail \Delta, \varphi}{\Gamma, \neg\varphi \rightarrowtail \Delta}\neg\textit{Gauche} \qquad \frac{\Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta, \neg\varphi}\neg\textit{Droite}$$

$$\frac{\Gamma, \varphi[x/y] \rightarrowtail \Delta}{\Gamma, \exists x.\varphi \rightarrowtail \Delta}\exists\textit{Gauche} \qquad \frac{\Gamma \rightarrowtail \Delta, \varphi[x/t]}{\Gamma \rightarrowtail \Delta, \exists x.\varphi}\exists\textit{Droite}$$

où la règle $\exists$Gauche vérifie la condition $eigenvariable$ (x n'est pas libre dans Γ, Δ).

$$\frac{\Gamma \rightarrowtail \Delta, \varphi \quad \Gamma', \varphi \rightarrowtail \Delta'}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'}\textit{Cut}$$

Dans la règle de coupure, φ est appelée **formule de coupure**. *Nous utiliserons la notion de* **formule principale** *pour une règle r de la manière suivante :*

- *$\neg\varphi$ est principale par rapport à $\neg$Gauche et $\neg$Droite,*
- *$\varphi \vee \varphi'$ est principale par rapport à $\vee$Gauche et $\vee$Droite,*
- *$\exists x.\varphi$ est principale par rapport à $\exists$Gauche et $\exists$Droite,*

3.3.3.2 Deux règles supplémentaires

Avant de commencer, nous allons ajouter deux règles à l'ensemble des règles d'inférences R_{LK} de notre système formel. Quand nous regardons les preuves du résultat d'élimination des coupures dans la littérature [Gal86, Kle52], nous constatons que deux lemmes intermédiaires apparaissent. Nous nous apercevons, en regardant de plus près, que ces deux lemmes peuvent être remplacés par des transformations élémentaires d'arbres de preuve. Mais pour cela, il faut ajouter deux règles supplémentaires.

– La règle de **substitution** :

$$\frac{\Gamma \rightarrowtail \Delta}{\Gamma_\sigma \rightarrowtail \Delta_\sigma}\ Subst$$

avec $\sigma : V \rightarrow T_\Sigma(V)$ une substitution et $\Gamma_\sigma = \varphi_{1_\sigma}, \ldots, \varphi_{n_\sigma}$ lorsque $\Gamma = \varphi_1, \ldots, \varphi_n$.
φ_{i_σ} signifie que l'on applique la substitution σ sur la formule φ_i.

– La règle d'**affaiblissement** :

$$\frac{\Gamma \rightarrowtail \Delta}{\Gamma' \rightarrowtail \Delta'}\ Aff \qquad \text{avec :} \quad \begin{array}{ll} & \Gamma' = \Gamma, \Phi \\ & \Delta' = \Delta, \Psi \\ \text{ou} & \Gamma' = \Gamma \\ & \Delta' = \Delta, \Psi \\ \text{ou} & \Gamma' = \Gamma, \Phi \\ & \Delta' = \Delta \end{array}$$

Cette seule règle d'affaiblissement permet d'effectuer un affaiblissement à gauche, à droite, ou les deux simultanément. Il est normalement déjà traité implicitement dans le calcul des séquents étudié, mais nous en aurons besoin dans la suite.

Ces deux règles vont être ajoutées règles du calcul des séquents. Il faut donc vérifier si elle sont bien admissibles pour l'ensemble des règles R_{LK}.

Définition 3.3.7 *Soit $\frac{A}{B}r$ une règle d'inférence, et soit $\mathcal{R}$ un système de preuve qui ne contient pas r.*
r est **admissible** *pour $\mathcal{R}$ si lorsqu'on a une preuve de A à l'aide de $\mathcal{R}$ alors on a aussi une preuve de B à l'aide de $\mathcal{R}$.*

Nous appellerons R'_{LK} le système de règle résultant de l'ajout de ces deux règles à R_{LK} qui sont évidemment admissible pour le calcul des séquents

3.3.3.3 Modélisation dans notre cadre abstrait

Le résultat d'élimination des coupures (et, en particulier, la terminaison qui est la partie la plus difficile) peut être montré en utilisant des transformations élémentaires d'arbres de preuve que nous allons donner maintenant. Ces règles ont pour but de faire remonter la règle de coupure dans un arbre et de l'éliminer ensuite sous les axiomes. L'ensemble de règles transformations peut être divisé en plusieurs catégories :

1) La formule de coupure n'est pas principale dans au moins une des prémisses

Voici les transformations lorsqu'une règle $Droite$ du calcul des séquents ($\neg Droite$, $\vee Droite$ et $\exists Droite$) est appliquée sur la prémisse gauche de la règle Cut, puis le cas symétrique où une règle $Gauche$ ($\neg Gauche, \vee Gauche$ et $\exists Gauche$) est appliquée sur la prémisse de droite.

$$\dfrac{\dfrac{\Gamma_1 \rightarrowtail \Delta_1, \varphi}{\Gamma_1' \rightarrowtail \Delta_2, \varphi}\,@Droite \qquad \Gamma_2, \varphi \rightarrowtail \Delta_3}{\Gamma_1', \Gamma_2 \rightarrowtail \Delta_2, \Delta_3}\,Cut \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma_1 \rightarrowtail \Delta_1, \varphi \quad \Gamma_2, \varphi \rightarrowtail \Delta_3}{\Gamma_1', \Gamma_2 \rightarrowtail \Delta_1, \Delta_3}\,Cut}{\Gamma_1', \Gamma_2 \rightarrowtail \Delta_2, \Delta_2}\,@Droite$$

avec $@ \in \{\vee, \neg, \exists\}$ et $\Gamma_1' = \Gamma_1$ sauf pour la règle $\neg Droite$, $\Gamma_1' = \Gamma_1, \neg\psi$ pour toute formule ψ.

$$\dfrac{\Gamma_3 \rightarrowtail \Delta_2, \varphi \qquad \dfrac{\Gamma_1, \varphi \rightarrowtail \Delta_1}{\Gamma_2, \varphi \rightarrowtail \Delta_1'}\,@Gauche}{\Gamma_3, \Gamma_2 \rightarrowtail \Delta_1', \Delta_2}\,Cut \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma_3 \rightarrowtail \Delta_2, \varphi \quad \Gamma_1, \varphi \rightarrowtail \Delta_1}{\Gamma_3, \Gamma_1 \rightarrowtail \Delta_1', \Delta_2}\,Cut}{\Gamma_3, \Gamma_2 \rightarrowtail \Delta_1', \Delta_2}\,@Gauche$$

avec $@ \in \{\vee, \neg, \exists\}$ et $\Delta_1' = \Delta_1$ sauf pour la règle $\neg Gauche$, $\Delta_1' = \Delta_1, \neg\psi$ pour toute formule ψ.

2) La formule de coupure est principale

Les règles juste au dessus de la coupure Cut utilise la formule de coupure comme formule principale. Dans ces cas là, nous avons les règles de transformation suivantes :

$$\dfrac{\dfrac{\Gamma, \varphi \rightarrowtail \Delta \quad \Gamma, \varphi' \rightarrowtail \Delta}{\Gamma, \varphi \vee \varphi' \rightarrowtail \Delta}\,\vee Gauche \qquad \dfrac{\Gamma' \rightarrowtail \Delta', \varphi, \varphi'}{\Gamma' \rightarrowtail \Delta', \varphi \vee \varphi'}\,\vee Droite}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'}\,Cut \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma, \varphi \rightarrowtail \Delta \qquad \Gamma' \rightarrowtail \Delta', \varphi, \varphi'}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta', \varphi'}\, Cut \qquad \Gamma, \varphi' \rightarrowtail \Delta}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'}\, Cut$$

$$\dfrac{\dfrac{\Gamma \rightarrowtail \Delta, \varphi}{\Gamma, \neg\varphi \rightarrowtail \Delta}\, \neg Gauche \qquad \dfrac{\Gamma', \varphi \rightarrowtail \Delta'}{\Gamma' \rightarrowtail \Delta', \neg\varphi}\, \neg Droite}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'}\, Coupure \rightsquigarrow$$

$$\dfrac{\Gamma \rightarrowtail \Delta, \varphi \;\; \Gamma', \varphi \rightarrowtail \Delta'}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'}\, Coupure$$

$$\dfrac{\dfrac{\Gamma, \varphi[x/y] \rightarrowtail \Delta}{\Gamma, \exists x.\varphi \rightarrowtail \Delta}\, \exists Gauche \qquad \dfrac{\Gamma' \rightarrowtail \Delta', \varphi[x/t]}{\Gamma' \rightarrowtail \Delta', \exists x.\varphi}\, \exists Droite}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'}\, Cut \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma, \varphi[x/y] \rightarrowtail \Delta}{\Gamma[y/t], \varphi[x/y][y/t] \rightarrowtail \Delta[y/t]}\, Subst \qquad \Gamma' \rightarrowtail \Delta', \varphi[x/t]}{\Gamma, \Gamma' \rightarrowtail \Delta, \Delta'}\, Cut$$

3) L'une des prémisses de la règle Cut est un axiome et l'autre n'en est pas un

$$\dfrac{\dfrac{}{\Gamma, \varphi \rightarrowtail \Delta, \varphi}\, Ax. \qquad \Gamma', \varphi \rightarrowtail \Delta'}{\Gamma, \varphi, \Gamma' \rightarrowtail \Delta, \Delta'}\, Cut \quad \rightsquigarrow \quad \dfrac{\Gamma', \varphi \rightarrowtail \Delta'}{\Gamma, \varphi, \Gamma' \rightarrowtail \Delta, \Delta'}\, Aff$$

Si l'axiome se trouve sur la prémisse de droite, la transformation se fait de manière similaire :

$$\dfrac{\Gamma \rightarrowtail \Delta, \varphi \qquad \dfrac{}{\Gamma', \varphi \rightarrowtail \Delta', \varphi}\, Ax.}{\Gamma, \Gamma' \rightarrowtail \Delta, \varphi, \Delta'}\, Cut \quad \rightsquigarrow \quad \dfrac{\Gamma \rightarrowtail \Delta, \varphi}{\Gamma, \Gamma' \rightarrowtail \Delta, \varphi, \Delta'}\, Aff$$

4) Les règles $Subst$ et Aff ont une prémisse qui n'est pas le Cut

Nous pouvons montrer facilement que les règles $Subst$ et Aff remontent sur toutes les règles $@Gauche$ et $@Droite$ avec $@ \in \{\neg, \vee, \exists\}$. Voici les règles de transformation :

La règle d'**affaiblissement** remonte sur toutes les règles de $@Gauche$ et $@Droite$.

1. *Aff* remonte sur $\neg Gauche$

$$\dfrac{\dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta}{\Gamma, \neg\Phi \rightarrowtail \Delta}\, \neg Gauche}{\Gamma', \neg\Phi \rightarrowtail \Delta'}\, Aff \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Delta}{\Gamma', \rightarrowtail \Phi, \Delta'}\, Aff}{\Gamma', \neg\Phi \rightarrowtail \Delta'}\, \neg Gauche$$

2. *Aff* remonte sur $\neg Droite$

$$\dfrac{\dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Delta}{\Gamma \rightarrowtail \neg\Phi, \Delta}\, \neg Droite}{\Gamma' \rightarrowtail \neg\Phi, \Delta'}\, Aff \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta}{\Gamma', \Phi \rightarrowtail \Delta'}\, Aff}{\Gamma' \rightarrowtail \neg\Phi, \Delta'}\, \neg Droite$$

3. *Aff* remonte sur $\vee Gauche$

$$\dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta \qquad \Gamma, \Phi' \rightarrowtail \Delta}{\Gamma, \Phi \vee \Phi' \rightarrowtail \Delta}\, \vee Gauche}{\Gamma', \Phi \vee \Phi' \rightarrowtail \Delta'}\, Aff \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta}{\Gamma', \Phi \rightarrowtail \Delta'}\, Aff \qquad \dfrac{\Gamma, \Phi' \rightarrowtail \Delta}{\Gamma', \Phi' \rightarrowtail \Delta'}\, Aff}{\Gamma', \Phi \vee \Phi' \rightarrowtail \Delta'}\, \vee Gauche$$

4. *Aff* remonte sur $\vee Droite$

$$\dfrac{\dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Phi', \Delta}{\Gamma \rightarrowtail \Phi \vee \Phi', \Delta}\, \vee Droite}{\Gamma' \rightarrowtail \Phi \vee \Phi', \Delta'}\, Aff \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Phi', \Delta}{\Gamma' \rightarrowtail \Phi, \Phi', \Delta'}\, Aff}{\Gamma' \rightarrowtail \Phi \vee \Phi', \Delta'}\, \vee Droite$$

5. *Aff* remonte sur $\exists Gauche$

$$\dfrac{\dfrac{\dfrac{\Gamma, \Phi[x/y] \rightarrowtail \Delta}{\Gamma, \exists x.\Phi \rightarrowtail \Delta}\, \exists Gauche}{\Gamma', \exists x.\Phi \rightarrowtail \Delta'}\, Aff \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\Gamma, \Phi[x/y] \rightarrowtail \Delta}{\Gamma', \Phi[x/y] \rightarrowtail \Delta'}\, Aff}{\Gamma', \exists x.\Phi \rightarrowtail \Delta'}\, \exists Gauche$$

6. *Aff* remonte sur $\exists Droite$

$$\dfrac{\dfrac{\dfrac{\Gamma \rightarrowtail \Delta, \Phi[x/t]}{\Gamma \rightarrowtail \exists x.\Phi, \Delta}\, \exists Droite}{\Gamma' \rightarrowtail \exists x.\Phi, \Delta'}\, Aff \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\Gamma \rightarrowtail \Delta, \Phi[x/t]}{\Gamma' \rightarrowtail \Phi[x/t], \Delta'}\, Aff}{\Gamma' \rightarrowtail \exists x.\Phi, \Delta'}\, \exists Droite$$

La règle de **substitution** remonte sur toutes les règles sauf sur les règles Cut et Aff.

1. ***Subst*** remonte sur $\vee Gauche$

$$\dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta \qquad \Gamma, \Phi' \rightarrowtail \Delta}{\Gamma, \Phi \vee \Phi' \rightarrowtail \Delta}\ \vee Gauche}{\Gamma_\sigma, (\Phi \vee \Phi')_\sigma \rightarrowtail \Delta_\sigma}\ Subst \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta}{\Gamma_\sigma, \Phi_\sigma \rightarrowtail \Delta_\sigma}\ Subst \qquad \dfrac{\Gamma, \Phi' \rightarrowtail \Delta}{\Gamma_\sigma, \Phi'_\sigma \rightarrowtail \Delta_\sigma}\ Subst}{\Gamma_\sigma, \Phi_\sigma \vee \Phi'_\sigma \rightarrowtail \Delta_\sigma}\ \vee Gauche$$

2. ***Subst*** remonte sur $\vee Droite$

$$\dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Phi', \Delta}{\Gamma \rightarrowtail \Phi \vee \Phi', \Delta}\ \vee Droite}{\Gamma_\sigma \rightarrowtail (\Phi \vee \Phi')_\sigma, \Delta_\sigma}\ Subst \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Phi', \Delta}{\Gamma_\sigma \rightarrowtail \Phi_\sigma, \Phi'_\sigma, \Delta_\sigma}\ Subst}{\Gamma_\sigma \rightarrowtail \Phi_\sigma \vee \Phi'_\sigma, \Delta_\sigma}\ \vee Droite$$

3. ***Subst*** remonte sur $\neg Gauche$

$$\dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta}{\Gamma, \neg\Phi \rightarrowtail \Delta}\ \neg Gauche}{\Gamma_\sigma, (\neg\Phi)_\sigma \rightarrowtail \Delta_\sigma}\ Subst \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Delta}{\Gamma_\sigma \rightarrowtail \Phi_\sigma, \Delta_\sigma}\ Subst}{\Gamma_\sigma, \neg\Phi_\sigma \rightarrowtail \Delta_\sigma}\ \neg Gauche$$

4. ***Subst*** remonte sur $\neg Droite$

$$\dfrac{\dfrac{\Gamma \rightarrowtail \Phi, \Delta}{\Gamma \rightarrowtail \neg\Phi, \Delta}\ \neg Droite}{\Gamma_\sigma \rightarrowtail (\neg\Phi)_\sigma, \Delta_\sigma}\ Subst \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma, \Phi \rightarrowtail \Delta}{\Gamma_\sigma, \Phi_\sigma \rightarrowtail \Delta_\sigma}\ Subst}{\Gamma_\sigma \rightarrowtail \neg\Phi_\sigma, \Delta_\sigma}\ \neg Droite$$

5. ***Subst*** remonte sur $\exists Gauche$

$$\dfrac{\dfrac{\Gamma, \Phi[y/x] \rightarrowtail \Delta}{\Gamma, \exists x.\Phi \rightarrowtail \Delta}\ \exists Gauche}{\Gamma_\sigma, (\exists x.\Phi)_\sigma \rightarrowtail \Delta_\sigma}\ Subst \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma \rightarrowtail \Phi[y/x], \Delta}{\Gamma_\sigma \rightarrowtail (\Phi[y/x])_\sigma, \Delta_\sigma}\ Subst}{\Gamma_\sigma, (\exists x.\Phi)_\sigma \rightarrowtail \Delta_\sigma}\ \exists Gauche$$

(y variable non-libre dans Γ et Δ)

6. ***Subst*** remonte sur $\exists Droite$

$$\dfrac{\dfrac{\Gamma \rightarrowtail \Phi[t/x], \Delta}{\Gamma \rightarrowtail \exists x.\Phi, \Delta}\ \exists Droite}{\Gamma_\sigma \rightarrowtail (\exists x.\Phi)_\sigma, \Delta_\sigma}\ Subst \quad \rightsquigarrow \quad \dfrac{\dfrac{\Gamma, \Phi[t/x] \rightarrowtail \Delta}{\Gamma_\sigma, (\Phi[t/x])_\sigma \rightarrowtail \Delta_\sigma}\ Subst}{\Gamma_\sigma \rightarrowtail (\exists x.\Phi)_\sigma, \Delta_\sigma}\ \exists Droite$$

5) Élimination des règles Cut, $Subst$ et Aff sous les axiomes

1. La règle de ***substitution*** s'annule sous un ***Axiome*** :

$$\dfrac{\dfrac{}{\Gamma, \Phi \rightarrowtail \Phi, \Delta}\, Ax}{\Gamma_\sigma, \Phi_\sigma \rightarrowtail \Phi_\sigma, \Delta_\sigma}\, Subst \quad \rightsquigarrow \quad \dfrac{}{\Gamma_\sigma, \Phi_\sigma \rightarrowtail \Phi_\sigma, \Delta_\sigma}\, Ax$$

2. La règle d'***Affaiblissement*** s'annule sous un ***Axiome*** :

$$\dfrac{\dfrac{}{\Gamma, \Phi \rightarrowtail \Phi, \Delta}\, Ax}{\Gamma, \Phi, \Psi \rightarrowtail \Phi, \Psi', \Delta}\, Aff \quad \rightsquigarrow \quad \dfrac{}{\Gamma, \Phi, \Psi \rightarrowtail \Phi, \Psi', \Delta}\, Ax$$

3. Avec deux axiomes dans les prémisses du Cut, la conclusion est elle-même un axiome. En effet la coupure s'effectuant au maximum sur une seule formule principale d'un des axiomes, il reste un axiome dans la conclusion.

$$\dfrac{\dfrac{}{\Gamma, \Phi \rightarrowtail \Phi, \Xi, \Delta}\, Axiome \qquad \dfrac{}{\Gamma', \Psi, \Xi \rightarrowtail \Psi, \Delta'}\, Axiome}{\Gamma, \Gamma', \Phi, \Psi \rightarrowtail \Phi, \Psi, \Delta, \Delta'}\, Coupure \quad \rightsquigarrow$$

$$\dfrac{}{\Gamma, \Gamma', \Phi, \Psi \rightarrowtail \Phi, \Psi, \Delta, \Delta'}\, Axiome$$

(ou)

$$\dfrac{\dfrac{}{\Gamma, \Phi \rightarrowtail \Phi, \Xi, \Delta}\, Axiome \qquad \dfrac{}{\Gamma', \Xi \rightarrowtail \Xi, \Delta'}\, Axiome}{\Gamma, \Gamma', \Phi \rightarrowtail \Phi, \Xi, \Delta, \Delta'}\, Coupure \quad \rightsquigarrow$$

$$\dfrac{}{\Gamma, \Gamma', \Phi \rightarrowtail \Phi, \Xi, \Delta, \Delta'}\, Axiome$$

(ou)

$$\dfrac{\dfrac{}{\Gamma, \Xi \rightarrowtail \Xi, \Delta}\, Axiome \qquad \dfrac{}{\Gamma', \Xi \rightarrowtail \Xi, \Delta'}\, Axiome}{\Gamma, \Gamma', \Xi \rightarrowtail \Xi, \Delta, \Delta'}\, Coupure \quad \rightsquigarrow$$

$$\dfrac{}{\Gamma, \Gamma', \Xi \rightarrowtail \Xi, \Delta, \Delta'}\, Axiome$$

L'ensemble des règles de transformation données est composé de toutes les règles permettant de faire remonter les règles de coupure, d'affaiblissement, et de substitution dans un arbre. L'arbre de preuve obtenu ne contiendra donc aucune instance de $@Gauche$ et $@Droite$ au-dessus de ces règles. Les règles Cut, Aff

et $Subst$ se trouvent alors directement sous les axiomes, et comme le montre les règles du point 5), elles s'éliminent toutes pour obtenir un arbre uniquement constitué d'axiomes et d'instances de $@Gauche$ et $@Droite$. La forme normale obtenue après l'application de ces transformations est nécessairement celle associée à un arbre de preuve du calcul des séquents sans la règle de coupure.

Avec la figure 3.2, nous représentons la procédure en deux étapes (même si en théorie elle se font simultanément). La première, correspondant à la remontée des règles Cut, Aff et $Subst$ sur toutes les autres (points 1),2),3),4)) et la seconde, correspondant à l'élimination de ces trois règles (point 5)).

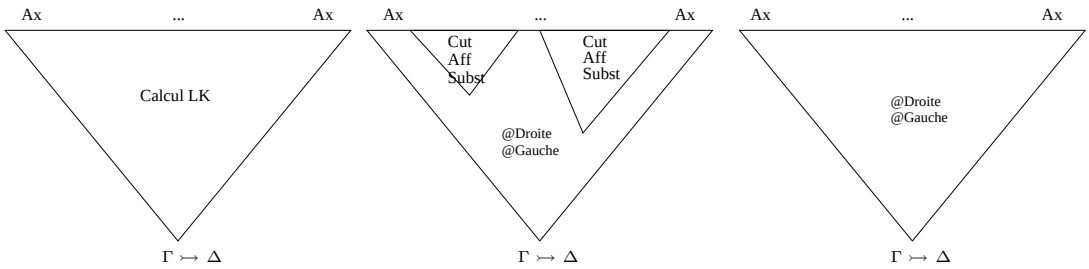


FIG. 3.2 – Deux étapes pour l'élimination des coupures

3.3.3.4 Vérification des conditions

Il est facile de montrer que nos transformations élémentaires vérifient les conditions pour la normalisation forte de la définition 3.2.1 page 73. grâce à l'ordre bien-fondé $\preceq$ suivant :

$$\forall @ \in \{\vee, \neg, \exists\},\ Cut \ \succ\ Aff \sim Subst \ \succ\ @Gauche \sim @Droite \ \sim\ Ax.$$

et

$$\frac{\Gamma_1 \rightarrowtail \Delta_1, \varphi_1 \quad \Gamma'_1, \varphi_1 \rightarrowtail \Delta'_1}{\Gamma_1, \Gamma'_1 \rightarrowtail \Delta_1, \Delta'_1}\ Cut \quad \succ \quad \frac{\Gamma_2 \rightarrowtail \Delta_2, \varphi_2 \quad \Gamma'_2, \varphi_2 \rightarrowtail \Delta'_2}{\Gamma_2, \Gamma'_2 \rightarrowtail \Delta_2, \Delta'_2}\ Cut$$

$$\Leftrightarrow |\varphi_2| < |\varphi_1|$$

avec $|\varphi|$ la profondeur de la formule telle que $|\varphi| = sup_k(1 + |\varphi_k|)$ si les φ_i sont les sous-formules directes de φ. Cette dernière équivalence signifie que l'on classe les instances de la règle Cut, par l'ordre de précédence $\preceq$, en fonction de la taille de la formule de coupure. Si deux formules α_1, α_2 sont de la même taille

$|\alpha_1| = |\alpha_2|$ alors deux règles de coupure, utilisant respectivement α_1 et α_2 comme formule de coupure, seront de rang équivalent dans l'ordre $\preceq$.

Comme nous pouvons le vérifier facilement, toutes les règles de transformation pour l'élimination des coupures sont de la forme :

$$\frac{\iota_1 \ \dots \ \iota_n}{\varphi}\iota \rightsquigarrow \frac{\iota'_1 \ \dots \ \iota'_m}{\varphi}\iota'$$

telles que $\iota \succ \iota'$ et telles que pour tout $1 \leq i \leq m$, $\iota'_i \in F \cup \bigcup_{r \in R} r$. Elles vérifient donc toutes les conditions pour la normalisation forte. Ainsi, à l'aide du théorème 3.2.1, nous pouvons dire que la procédure d'élimination des coupures présentée ici termine et est fortement normalisante.

3.4 Conditions et théorème pour la normalisation faible

De la même manière que dans la section 3.2, cette section est dédiée à un théorème de normalisation faible sur les transformations d'arbres de preuve, et cela pour n'importe quel système formel. Dans beaucoup de situations, la procédure de normalisation forte fonctionne, mais parfois certaines règles de transformations peuvent poser des problèmes. C'est le cas pour les règles de la forme :

$$(\iota_1, \dots, \iota_n, \varphi)_\iota \rightsquigarrow \pi$$

telle qu'il existe des sous-arbres $\pi' = (\iota'_1, \dots, \iota'_m, \varphi')_{\iota'}$ de π avec $\iota'_j \in F \cup \bigcup_{r \in R} r$ pour tout $j, 1 \leq j \leq m$ satisfaisant :

- $\iota \sim \iota'$
- $\mathcal{LM}(\iota'_j) = \mathcal{LM}(\iota_1, \dots, \iota_n, \varphi)_\iota$

Ce type de situations empêche parfois l'utilisation des techniques standards pour prouver la terminaison (le RPO) et donc d'obtenir un résultat de normalisation forte.

Par exemple, c'est le cas du calcul des séquents pour lequel on considère la règle de coupure suivante :

$$\frac{\Gamma \rightarrowtail \Delta, \varphi \quad \Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta}\text{Coupure}$$

Cependant, avec une telle règle de coupure, nous avons besoin de la règle structurelle d'affaiblissement pour construire la règle de transformation suivante :

$$\textbf{R1}\quad \dfrac{\dfrac{\dfrac{\Gamma_1 \rightarrowtail \Delta_1, \varphi}{\Gamma'_1 \rightarrowtail \Delta'_1, \varphi}\ @Droite \qquad \Gamma'_1, \varphi \rightarrowtail \Delta'_1}{\Gamma'_1 \rightarrowtail \Delta'_1}\ Coupure \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \rightarrowtail \Delta_1, \varphi}{\Gamma_1, \Gamma'_1 \rightarrowtail \Delta_1, \Delta'_1, \varphi}\ Aff. \qquad \dfrac{\Gamma'_1, \varphi \rightarrowtail \Delta'_1}{\Gamma_1, \Gamma'_1, \varphi \rightarrowtail \Delta_1, \Delta'_1}\ Aff.}{\Gamma_1, \Gamma'_1 \rightarrowtail \Delta'_1, \Delta_1}\ Coupure}{\Gamma'_1 \rightarrowtail \Delta'_1}\ @Droite$$

Pour cette règle de transformation, seul diminue le nombre d'occurrences des règles $@Droite$ et $@Gauche$ (c'est-à-dire les règles qui n'appartiennent pas à l'ensemble $Elim$ défini ci-dessous) qui apparaissent au-dessus de la règle de coupure. En appliquant une stratégie qui élimine les arbres de preuves maximaux (voir définition ci-dessous) les plus proches des feuilles, cela suffit pour obtenir un résultat de normalisation faible.

Définition 3.4.1 (Arbre de preuve maximal) *Soit $\rightsquigarrow$ une procédure de transformations. Un arbre de preuve $\pi = (\pi_1, \ldots, \pi_n, \varphi)_\iota$ est dit* **maximal** *pour $\rightsquigarrow$ si et seulement si pour tout i, $1 \leq i \leq n$, chaque π_i est un arbre de preuve normalisé, mais où π ne l'est pas.*

Commençons par définir, pour une procédure de transformation donnée, l'ensemble $Elim$ de toutes les règles qui remonteront au moins une fois, parmi les règles de transformation, au-dessus des d'autres règles.

Définition 3.4.2 *Soit $\rightsquigarrow_{\mathcal{T}}$ une procédure de transformation. L'ensemble des règles $Elim$ est défini de la manière suivante :*

$$Elim = \{\iota \mid \exists (\iota_1, \ldots, \iota_n, \varphi)_\iota \rightsquigarrow \pi\}$$

La règle de Coupure dans l'élimination des coupures fait évidemment partie de cette ensemble, ainsi que les règles d'Affaiblissement, et de Substitution. Si une règle ι n'appartient pas à $Elim$, cela signifie qu'il n'existe pas de règles de transformation de la forme $(\iota_1, \ldots, \iota_n, \varphi)_\iota \rightsquigarrow \pi$. Pour l'élimination des coupures, toutes les instances de $@Gauche$ et $@Droite$ avec $@ \in \{\wedge, \neg, \exists\}$ font partie de l'ensemble $Elim$.

Nous parlerons également de la longueur d'une preuve qui nous donne le nombre d'instances de règles n'appartenant pas à $Elim$ dans une preuve π.

Définition 3.4.3 *Soit $\pi = (\pi_1, \ldots, \pi_n, \varphi)_\iota$ un arbre de preuve. La* **longueur** *de la preuve π, notée $|\pi|$, est définie inductivement de la manière suivante :*

– *si* π *est une feuille alors*
$$|\pi| = 0$$
– *si* $\iota \in Elim$ *alors*
$$|\pi| = \sum_i |\pi_i|$$
– *si* $\iota \notin Elim$ *alors*
$$|\pi| = \sum_i |\pi_i| + 1$$

De la même manière, les notions de rang et de mesure nous permettront de comparer les arbres de preuves.

Tous les ensembles bien-fondés sont isomorphiques à un unique ordinal. Notons $d : (\bigcup_{r \in R} r, \succeq) \to \alpha$ cet isomorphisme où α est un ordinal.

Définition 3.4.4 *Soit* π *une preuve, et soit* $\pi' = (\pi_1, \ldots, \pi_n, \varphi)_\iota$ *une sous-preuve de* π *avec* $\iota \in Elim$*.*

Le **rang** *de* ι *dans* π*, noté* $rk_\pi(\iota)$*, est égal à* $d(\iota) + |\pi'|$*.*

Le rang de ι est le nombre d'instances de règles qui ne sont pas dans $Elim$ et qui apparaissent au-dessus de ι dans π'.

Définition 3.4.5 *La* **mesure** *d'une preuve* π*, notée* $meas(\pi)$*, est égale à*
$$\sum_\iota rk_\pi(\iota)$$
où ι *couvre toutes les règles d'inférence de* π *qui appartiennent à* $Elim$*.*

Définition 3.4.6 (Conditions pour la normalisation faible) *Soit* $\mathcal{S} = (A, F, R)$ *un système formel. Soit* $\mathcal{T}$ *un système de transformation, et* $\preceq$ *un ordre bien-fondé sur l'ensemble des règles d'inférences* R*.*

Un procédure de transformation $\rightsquigarrow_\mathcal{T}$ *vérifie les* **conditions pour la normalisation faible** *si pour toute règle de transformation* $(\iota_1, \ldots, \iota_n, \varphi)_\iota \rightsquigarrow \pi \in \mathcal{T}$ *:*

1. *il existe* i*,* $1 \leq i \leq n$*, tel que* $\iota_i \notin Elim$*,*
2. *le multi-ensemble des feuilles* π *est inclus dans celui de* $(\iota_1, \ldots, \iota_n, \varphi)_\iota$ *(on notera* $\mathcal{LM}(\pi) \subseteq \mathcal{LM}((\iota_1, \ldots, \iota_n, \varphi)_\iota)$*),*
3. *pour toute occurrence de règle* ι' *apparaissant dans* pi*,* $\iota \succeq \iota'$*,*
4. *si* π *est de la forme* $(\pi_1, \ldots, \pi_m, \varphi)_{\iota'}$ *alors toute les instances de règles de* π *différentes de* ι' *appartiennent à* $Elim$*.*

Il est facile de vérifier que toutes ces conditions sont vérifiées pour **R1**. Nous verrons que toutes les autres règles pour l'élimination des coupures vérifient ces conditions.

Nous pouvons maintenant donner notre théorème de normalisation faible.

Théorème 3.4.1 *Toute procédure de transformation $\rightsquigarrow_{\mathcal{T}}$ satisfaisant les conditions de la définition 3.4.6 est faiblement normalisante.*

Preuve La preuve de ce théorème est obtenue en généralisant la preuve de Tait [Tai89].

Nous allons le prouver en montrant que l'application de n'importe qu'elle règle de transformation d'arbre de preuve n'augmente pas la mesure des preuves (voir définition 3.4.5). Cette preuve est construite en deux étapes. Premièrement, nous montrerons que pour tout arbre de preuve maximal π transformé en $\overline{\pi}$ par une règle de transformation nous avons $meas(\overline{\pi}) < meas(\pi)$. Puis, nous montrerons que ce résultat peut être étendu à une preuve quelconque en suivant la stratégie qui consiste à réduire les sous-arbres maximaux.

1. Soit $\pi = (\pi_1, \ldots, \pi_n, \varphi)_\iota$ un arbre de preuve maximal. Soit $(\iota_1, \ldots, \iota_n, \varphi)_\iota \rightsquigarrow \pi'$ une règle de transformation. Cette règle transforme π en $\overline{\pi}$. Comme π est un arbre de preuve maximal, pour toute sous-preuve $\pi'' = (\pi''_1, \ldots, \pi''_p, \varphi')_{\iota''}$ de $\overline{\pi}$ telle que $\iota'' \in Elim$, cela signifie que ι'' apparaît dans π', la partie gauche de la règle de transformation. Par la condition 4 de la définition 3.4.6, nous savons qu'aucune instance de règle, n'appartenant pas à $Elim$, n'a été introduite dans π'' au-dessus de ι''. De plus, par la condition 1, nous savons qu'il y a au moins une occurrence d'une instance de règle n'appartenant pas à $Elim$ qui n'apparaît plus dans π'', et par la condition 2, les autres occurrences de ces règles sont déjà dans π. Ainsi nous pouvons conclure que $|\pi''| \leq |\pi| - 1$. Finalement, par la condition 3, nous avons $\iota \succeq \iota''$. Donc, $d(\iota'') + |\pi''| \leq d(\iota) + |\pi| - 1$.
2. Soit π un arbre de preuve qui n'est ni en forme normale, ni maximal. Soit $\pi' = (\pi'_1, \ldots, \pi'_n, \varphi)_{\iota'}$ un sous-arbre de preuve de π qui n'est pas maximal et tel que $\iota \in Elim$. Toutes les règles de transformation qui peuvent être appliquées à π sont appliquées soit sur un sous-arbre maximal π'' de π (qui n'est pas π'), soit sur un sous-arbre maximal de π'. Dans le premier cas, $rk_\pi(\iota') = rk_{\overline{\pi}}(\iota')$ où $\overline{\pi}$ est la preuve obtenue à partir de π après application de la règle de transformation que l'on considère. Dans le second cas, par les conditions 1 et 4, la règle de transformation a éliminé au moins une occurrence d'une règle n'appartenant pas à $Elim$, et en a ajouté au plus une. Ainsi nous avons, $rk_\pi(\iota') \geq rk_{\overline{\pi}}(\iota')$.

Dans les deux cas que nous venons de voir, nous pouvons dire que $meas(\pi) > meas(\overline{\pi})$, et donc en conclure que la procédure de transformation termine.

★

3.5 Exemple de normalisation faible

3.5.1 Un calcul des séquents un peu différent

Nous pouvons reprendre l'exemple de l'élimination des coupures vue dans la section 3.3.3 page 82 de ce chapitre, en considérant cette fois la règle de coupure suivante :

$$\frac{\Gamma \rightarrowtail \Delta, \varphi \quad \Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta}\text{Coupure}$$

Avec une telle règle, les conditions pour la normalisation forte ne peuvent plus être vérifiées pour toutes les règles de transformation, comme nous l'avons vu dans la section précédente pour la règle **R1**.

3.5.2 Les règles de transformation

Toutes les transformations qui ne font pas intervenir la règle de coupure ne vont pas changer. Par contre il va falloir adapter les autres. Voici ces règles de transformations adaptées en suivant la méthode déjà vue dans la section 3.3.3.

1) La formule de coupure n'est pas principale dans au moins une des prémisses

Voici les transformations lorsqu'une règle $Droite$ du calcul des séquents ($\neg Droite$, $\vee Droite$ et $\exists Droite$) est appliquée sur la prémisse gauche de la règle $Coupure$. On retrouve la règle **R1** déjà vue dans la section précédente. Le cas symétrique où l'on a une règle Gauche se résout de manière similaire.

$$\textbf{R1}\quad \frac{\dfrac{\dfrac{\Gamma_1 \rightarrowtail \Delta_1, \varphi}{\Gamma'_1 \rightarrowtail \Delta'_1, \varphi}\,@Droite \qquad \Gamma'_1, \varphi \rightarrowtail \Delta'_1}{\Gamma'_1 \rightarrowtail \Delta'_1}\,Coupure}{} \quad \rightsquigarrow$$

$$\frac{\dfrac{\dfrac{\Gamma_1 \rightarrowtail \Delta_1, \varphi}{\Gamma_1, \Gamma'_1 \rightarrowtail \Delta_1, \Delta'_1, \varphi}\,Aff. \qquad \dfrac{\Gamma'_1, \varphi \rightarrowtail \Delta'_1}{\Gamma_1, \Gamma'_1, \varphi \rightarrowtail \Delta_1, \Delta'_1}\,Aff.}{\Gamma_1, \Gamma'_1 \rightarrowtail \Delta'_1, \Delta_1}\,Coupure}{\Gamma'_1 \rightarrowtail \Delta'_1}\,@Droite$$

avec $@ \in \{\vee, \neg, \exists\}$ et $\Gamma'_1 = \Gamma_1$ sauf pour la règle $\neg Droite$, $\Gamma'_1 = \Gamma_1, \neg\psi$ pour toute formule ψ.

2) La formule de coupure est principale

Les règles juste au dessus de la coupure $Coupure$ utilise la formule de coupure comme formule principale. Dans ces cas là, nous avons les règles de transformation suivantes :

$$\dfrac{\dfrac{\Gamma, \varphi \rightarrowtail \Delta \;\; \Gamma, \varphi' \rightarrowtail \Delta}{\Gamma, \varphi \vee \varphi' \rightarrowtail \Delta}\vee Gauche \qquad \dfrac{\Gamma \rightarrowtail \Delta, \varphi, \varphi'}{\Gamma \rightarrowtail \Delta, \varphi \vee \varphi'}\vee Droite}{\Gamma \rightarrowtail \Delta} Coupure \rightsquigarrow$$

$$\dfrac{\dfrac{\dfrac{\Gamma, \varphi \rightarrowtail \Delta}{\Gamma, \varphi \rightarrowtail \Delta, \varphi'} Aff \qquad \Gamma \rightarrowtail \Delta, \varphi, \varphi'}{\Gamma \rightarrowtail \Delta, \varphi'} Coupure \qquad \Gamma, \varphi' \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta} Coupure$$

$$\dfrac{\dfrac{\Gamma \rightarrowtail \Delta, \varphi}{\Gamma, \neg\varphi \rightarrowtail \Delta}\neg Gauche \qquad \dfrac{\Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta, \neg\varphi}\neg Droite}{\Gamma \rightarrowtail \Delta} Coupure \rightsquigarrow$$

$$\dfrac{\Gamma \rightarrowtail \Delta, \varphi \;\; \Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta} Coupure$$

$$\dfrac{\dfrac{\Gamma, \varphi[x/y] \rightarrowtail \Delta}{\Gamma, \exists x.\varphi \rightarrowtail \Delta}\exists Gauche \qquad \dfrac{\Gamma \rightarrowtail \Delta, \varphi[x/t]}{\Gamma \rightarrowtail \Delta, \exists x.\varphi}\exists Droite}{\Gamma \rightarrowtail \Delta} Coupure \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma, \varphi[x/y] \rightarrowtail \Delta}{\Gamma[y/t], \varphi[x/y][y/t] \rightarrowtail \Delta[y/t]} Subst \qquad \Gamma \rightarrowtail \Delta, \varphi[x/t]}{\Gamma \rightarrowtail \Delta} Coupure$$

3) L'une des prémisses de la règle $Coupure$ est un axiome

$$\dfrac{\dfrac{}{\Gamma \rightarrowtail \Delta, \varphi} Ax. \qquad \Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta} Coupure \quad \rightsquigarrow \quad \Gamma \rightarrowtail \Delta$$

si il existe une formule φ' telle que $\varphi' \in \Gamma$ et $\varphi' \in \Delta$,
(ou)

$$\dfrac{\dfrac{}{\Gamma \rightarrowtail \Delta, \varphi} Ax. \qquad \Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta} Coupure \quad \rightsquigarrow \quad \dfrac{\Gamma, \varphi \rightarrowtail \Delta}{\Gamma \rightarrowtail \Delta} Aff$$

si $\varphi \in \Gamma$.

Le cas symétrique, lorsque l'axiome se trouve sur la prémisse de droite, se fait de manière similaire.

4) Les transformations ne changent pas par rapport à la section 3.3.3

5) Remplacé par le paragraphe 3) de cette section pour la partie concernant la règle de *Coupure*, **sinon les autres transformations ne changent pas.**

3.5.3 Vérification des conditions

Toutes ces transformations respectent les conditions pour la normalisation forte sauf la règle **R1** qui pose problème. Cependant, celle-ci respecte les conditions pour la normalisation faible de la définition 3.4.6. On peut vérifier très facilement que les autres transformations vérifient également les conditions pour la normalisation faible. Donc cette procédure de transformation d'arbres de preuve est faiblement normalisante.

Troisième partie

Test à partir de spécifications algébriques

Chapitre 4

État de l'art

Les méthodes pour tester des logiciels sont nombreuses et variées comme nous avons pu le voir dans l'introduction de ce manuscrit. L'approche que nous étudions dans les prochains chapitres est celle du test fonctionnel, qui consiste à tester un programme uniquement à partir de sa description, sans utiliser son code.

Dans ce chapitre, après une introduction sur le test fonctionnel, nous allons présenter plusieurs outils qui illustrent bien la problématique de la sélection de tests à partir de spécifications formelles axiomatiques.

4.1 Introduction

4.1.1 Le test fonctionnel

Notre approche pour tester un programme informatique s'inscrit dans le cadre du test fonctionnel. Cette pratique de vérification a pour caractéristique principale le fait que l'on ne se sert pas du code du programme (d'où le nom parfois utilisé de test "boîte noire"). Nous ne connaissons pas le contenu du code mais uniquement la relation entre ses entrées/sorties. Il s'oppose au test dit structurel, ou "boîte blanche", où l'on s'appuie sur ce contenu.

L'un des principaux avantages du test fonctionnel, par rapport au structurel, est qu'il utilise le même support (la spécification) pour sélectionner les entrées de test et les sorties attendues (c'est-à-dire résoudre le problème de l'oracle). Dans l'approche fonctionnelle, les services que doit rendre le programme sont décrits d'une manière ou d'une autre.

Les spécifications répondent à ce besoin de description de propriétés attendues. Elle permettent de donner le comportement voulu pour un programme. C'est une idée ancienne [GG75] qui a été reprise de nombreuses fois comme nous allons le voir dans ce chapitre. Cette approche a un gros avantage par rapport à

d'autre méthodes : même si le programme change, sa spécification reste la même. Les tests construits n'auront pas besoin d'être modifiés même après des phases de correction du code (non régression). En outre, quand on écrit la spécification on décrit le comportement d'une fonction pour des valeurs données, et on dispose ainsi du résultat attendu en fonction des données d'entrée. C'est cette donnée du résultat attendu qui facilite la résolution du problème de l'oracle.

Dans le cadre de cette thèse nous allons parler du test à partir de spécifications algébriques (voir chapitre III pour la présentation détaillée), mais il existe d'autres méthodes de test fonctionnel largement utilisées où la description du comportement du programme est donnée sous d'autres formes. Par exemple à l'aide d'automates, on parle alors de test de conformité [Tre92, Tre96, LY96] ; ou bien à l'aide d'une spécification écrite en B ou Z [LPU02] (pour tester des systèmes dans des valeurs aux limites).

4.1.2 La sélection

Dans cette section, les tests sont sélectionnés à partir d'une spécification du programme donnée sous forme axiomatique. L'idée première [BGM91] fut donc de prendre simplement les axiomes de la spécification, et de dire que ceux-ci sont des tests (à substitution près des variables par des termes clos). En effet, les axiomes sont des propriétés que doit vérifier le programme. Si l'on sélectionne des jeux de valeurs pour les variables présentes dans un axiome donné, nous obtenons des tests que l'on peut soumettre au programme. Par exemple, dans le cas de l'axiome $(Add2)$ de l'addition (voir exemple 1.4.1 page 36),

$$succ(x) + y = succ(x + y)$$

si l'on substitue 0 à x et $succ(0)$ à y, on obtient la formule close suivante :

$$succ(0) + succ(0) = succ(0 + succ(0))$$

Cette formule est un test pour l'opération d'addition $+$. En effet, il suffit d'exécuter successivement les deux expressions $succ(0) + succ(0)$ et $succ(0 + succ(0))$ et de récupérer les deux résultats respectivement v_1 et v_2. Si les valeurs v_1 et v_2 coïncident, le test est en succès, sinon le test est en échec. Cette idée est à l'origine des critères d'uniformité et de régularité [BGM91, Mar91a]. Le critère d'uniformité est couvert lorsque l'on sélectionne au moins un test pour chaque axiome (par exemple le test de $+$ donné précédemment). Le critère de régularité d'ordre k est couvert lorsque tous les termes d'un type s donné et de taille inférieure ou égale à k sont sélectionnés. Par exemple, l'application du critère de régularité d'ordre 3 sur les entiers de l'axiome précédent consiste à substituer x et y successivement par 0, $succ(0)$ et $succ(succ(0))$. De même, la régularité d'ordre 3 sur

une pile d'entiers p (voir exemple 1.4.3 page 38) consisterait à substituer p par $vide$, $empiler(x, vide)$, $empiler(y, empiler(x, vide))$ où x et y sont des entiers sur lesquels on peut appliquer un critère d'uniformité ou à nouveau un critère de régularité.

Idéalement, la validation complète d'un programme par rapport à une spécification peut être obtenue par le test, à condition de soumettre l'ensemble des tests du jeu exhaustif des tests (dans l'exemple qui précède, l'ensemble de toutes les substitutions de tous les axiomes). Ceci est bien sûr impossible à réaliser en pratique, car le plus souvent le jeu de tests exhaustif contient une infinité de tests ou au moins un ensemble impraticable. La sélection d'un ensemble de tests (très petit par rapport à l'ensemble exhaustif) est donc une nécessité. Cette sélection doit donc être effectuée le plus judicieusement possible, car elle détermine directement la qualité de la vérification réalisée lors de l'activité de test. Il existe deux grandes classes de méthodes de sélection : les méthodes aléatoires, et les méthodes par partitionnement.

– Dans le cas de **méthodes aléatoires** [CH00, BBG97], les tests sont choisis parmi l'ensemble de tous les tests possibles (donc des substitutions d'axiomes) en sélectionnant de manière aléatoire. Elles ont l'avantage d'être faciles à implémenter, et permettent de générer une grande quantité de tests sans difficulté. Cependant, l'un des inconvénients majeur des méthodes aléatoires est le cas où un sous-domaine a une probabilité très faible d'apparaître. Dans le cas des spécifications conditionnelles positives, supposons que l'on a un axiome de la forme :

  $est\_triee(y :: l) = true \wedge x < y = true \Rightarrow insert(x, y :: l) = x :: y :: l$

  avec x, y des variables de types $entier$, l une variable de type liste, est_triee un prédicat qui vérifie qu'une liste est triée, $::$ le constructeur de liste et $insert$ l'opération d'insertion d'un élément dans un liste triée (opération sous test). La génération de données de tests de manière aléatoire risque de poser des problèmes pour cet axiome. Mis à part pour de toutes petites listes, il est difficile de générer aléatoirement des listes triées parmi toutes les listes. De manière plus générale, il est souvent difficile de trouver de manière aléatoire des tests intéressants qui vérifient les prémisses des axiomes.

– Les méthodes qui vont nous intéresser dans cette thèse relèvent d'une démarche plus dirigée. On parle de **méthodes par partition** [BGM91, DF93, BW05c, KATP02]. Les tests sont sélectionnés à partir des axiomes dans le but de réaliser une partition, la plus judicieuse possible, de l'ensemble de tous les tests. On réalise un découpage en sous-domaines de l'ensemble de tous les cas de tests possibles, puis on choisit des tests dans chaque sous-domaine. Ces tests (au moins un par sous-domaine) pourront

être soumis au programme. Ces tests sont également générés de manière aléatoire, mais sur un domaine beaucoup plus petit puisque l'on a "découpé" le domaine de définition principal. Le test par partition ne pose pas le problème des méthodes aléatoires puisque le découpage, s'il est bien fait, couvrira tous les sous-domaines pertinents de la spécification. Pour ce faire, l'utilisateur défini lui même son découpage selon ses besoins. Ainsi, il peut découper (et donc tester) plus ou moins finement les différentes parties de la spécification selon qu'elles sont plus ou moins critiques. Dans le chapitre 6 de cette thèse nous introduirons nos propres méthodes de sélection de tests par partition.

4.1.3 Les outils

Plusieurs méthodes et outils de génération automatique de tests à partir de spécifications ont été développés. Nous allons présenter plusieurs de ces outils dans les sections suivantes. Ces outils fonctionnent tous plus ou moins selon les trois phases suivantes :

- Il faut tout d'abord écrire une spécification qui contient les propriétés que l'on veut tester. Le langage utilisé dépend des outils, mais ce sont le plus souvent des formules proches de la logique des prédicats du premier ordre.
- Ensuite vient une phase de sélection, où les outils génèrent un ensemble de tests construits en accord avec la spécification.
- Puis pour finir, une phase où ces tests sont soumis à un programme (écrit dans divers langages, le plus souvent fonctionnels) et où l'on vérifie qu'ils sont corrects (c'est-à-dire que la valeur calculée via la spécification n'est pas mise en échec par le programme).

Nous allons commencer par présenter l'outil HOL-TestGen, qui sélectionne des tests pour un programme (écrit en SML, ou en C) à partir d'une spécification écrite dans le langage HOL (un langage de spécification de haut niveau). Les tests sont sélectionnés à partir d'une partition du domaine de tous les tests.

Ensuite nous présenterons, l'outil QuickCheck qui s'intègre complètement dans le langage de programmation Haskell, avec une spécification écrite également en Haskell. Contrairement, à HOL-TestGen, QuickCheck est un outil qui génère les tests de manière aléatoire.

Pour finir, nous nous intéresserons à l'outil LOFT, plus ancien que les deux outils précédent, mais dont nous nous sommes inspirés pour décrire les méthodes de sélection de tests décrites dans cette thèse.

4.2 HOL-TestGen

HOL-TestGen est un outil développé par Achim D. Brucker et Burkhart Wolff [BW05a, BW05b, BW07]. Cet outil s'intègre à l'assistant de preuve Isabelle comme une nouvelle "logique". Ces nouveaux éléments permettent de générer des tests pour des programmes écrit en SML ou bien en C. C'est un outil qui utilise une *méthode par partition* pour sélectionner les tests à l'aide des axiomes de la spécification.

Dans un premier temps nous allons présenter Isabelle et la logique HOL utilisée par l'outil HOL-TestGen. Ensuite nous présenterons le fonctionnement général de HOL-TestGen, que nous illustrerons ensuite sur un exemple.

4.2.1 Isabelle/HOL

Isabelle est un assistant de preuve générique [Wen05]. De nombreuses logiques sont supportées par Isabelle, notamment la logique du premier ordre (FOL), la théorie des ensembles de Zermelo-Fränkel (ZF), et bien sûr HOL que les auteurs ont choisis pour développer HOL-TestGen.

HOL, qui signifie Higher-order logic en anglais, est une logique classique enrichie. Elle est plus expressive que la logique du premier ordre car elle combine un langage de programmation fonctionnelle typé (tels que CAML, SML, ou Haskell...) et les quantificateurs. HOL est le langage de spécification utilisé par l'outil HOL-TestGen. L'association entre Isabelle et HOL s'appelle Isabelle/HOL [NPW02].

Isabelle/HOL est utilisé par HOL-TestGen comme un environnement de calcul symbolique. Cet environnement, via HOL, dispose d'un ensemble important de théories comme par exemple les ensembles, les listes, les multi-ensembles, les relations d'ordres... C'est également un véritable langage de spécification qui permet de définir nos propres types et fonctions. Les manipulées par Isabelle/HOL sont de la forme $A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow A_{n+1}$ que l'on note aussi $[|A_1, \ldots, A_n|] \Rightarrow A_{n+1}$. Elles signifient qu'à partir des contraintes $A_1, \ldots, A_n$, on infère la conclusion A_{n+1}.

4.2.2 HOL-TestGen

HOL-TestGen est un outil de test qui permet

- d'écrire des spécifications (pour le test) en HOL, c'est-à-dire de **spécifier** des structures de données et des opérations (via HOL), ainsi que des expressions à tester.
- de partitionner l'espace d'entrée de ces expressions (en utilisant la spécification) et de générer ainsi des **tests abstraits** (en s'appuyant sur les

hypothèses d'uniformité et de régularité que nous étudierons plus en détail dans le prochain chapitre)
- de sélectionner automatiquement des **tests concrets** à partir des tests abstraits,
- de générer des scripts en SML pour **exécuter** ces tests concrets sur des programmes écrits en SML : ces scripts contiennent les données de test à soumettre au programme ainsi que le résultat attendu (calculé via la spécification). La soumission au programme se fait via un "harnais" de test qui est une sorte d'outillage pour les scripts.
- tester des implantations dans d'autres langages comme C.

Je vais seulement expliquer la partie génération de tests abstraits à l'aide d'un exemple, et simplement commenter la génération des tests concrets. En revanche, je ne parlerai pas de la génération de scripts.

4.2.3 Exemple

À titre d'exemple, nous allons tester une opération d'insertion dans un arbre binaire de recherche. Isabelle manipule des modules, que l'on appelle théories. Ces théories peuvent être définies à partir d'autres théories existantes. Dans HOL-TestGen, on importe une théorie appelée `Testing`, qui contient un ensemble important de structures de données prédéfinies, ainsi qu'un ensemble de fonctions pour générer des tests.

En mettant dans un seul fichier l'ensemble des expressions qui vont suivre, on obtient une séquence utilisable par HOL-TestGen. La théorie que l'on définit ici a pour nom `ABR` :

```
theory ABR = Testing:
```

Il est possible de spécifier des types comme dans un type somme d'un langage fonctionnel, par exemple le type `'a tree` des arbres binaires, où `'a` est un type générique.

```
datatype 'a tree = ET |  MKT 'a "'a tree" "'a tree"
```

où `ET` est l'arbre vide, et `MKT` le constructeur d'arbre avec deux fils et une racine de type `'a`.

Ensuite, il est possible de déclarer des opérations ou des prédicats en donnant leurs profils (c'est notre signature).

```
consts
  is_in  :: "('a::order) => 'a tree => bool"
  is_ord :: "('a::order) tree => bool"
  insert :: "('a::order) => 'a tree => 'a tree"
```

`is_in` est un prédicat qui vérifie qu'un élément est bien dans un arbre binaire.

`is_ord` est un prédicat qui vérifie si un arbre binaire est un arbre binaire de recherche.

`insert` est une opération qui insère un élément au bon endroit dans un arbre binaire de recherche.

Le type `('a ::order)` signifie que le type `'a` est muni d'une relation d'ordre (c'est-à-dire que l'on pourra utiliser les prédicats $<, >, \ldots$ sur ce type).

En respectant les profils des opérations on peut les spécifier récursivement à l'aide d'un certain nombre de formules (les axiomes) spécifiées en HOL.

```
primrec
"is_in k ET = False"
"is_in k (MKT n l r) = ((k = n) ⋁ (is_in k l) ⋁ (is_in k r))"

primrec
isord_base : "is_ord ET = True"
isord_rec : "is_ord (MKT n l r) = ((∀n'. is_in n' l -> n' < n) ⋀
                                   (∀n'. is_in n' r -> n < n') ⋀
                                   is_ord l ⋀ is_ord r)"

primrec
insert_base:  "insert x ET = MKT x ET ET"
insert_rec:   "insert x (MKT n l r) =
                  (if x=n
                   then MKT n l r
                   else if x<n
                        then MKT n (insert x l) r
                        else MKT n l (insert x r))"
```

Tout ce que nous venons de définir correspond à notre spécification. Les éléments syntaxiques sont faciles à interpréter conformément à l'intuition dénotée par les différents symboles ou identificateurs. Pour l'instant la syntaxe utilisée est celle d'Isabelle. Nous allons maintenant expliquer la partie test, dont voici un exemple pour la fonction d'insertion :

```
test_spec "prog x t = insert x t"
 apply(gen_test_cases 1 1 "prog")
  store_test_thm "test"
  gen_test_data "test"

thm test.test_data
```

Le mot clé `test_spec` spécifie ce que l'on veut tester (notre objectif de test). Ici, nous voulons vérifier que l'implantation `prog` de l'opération `insert` est correcte (c'est-à-dire renvoie la même chose que la spécification).

Voici ce que Isabelle répond après cette première étape.

```
> test_spec "prog x t = insert x t"

proof (prove): step 0

fixed variables: prog, x, t

goal (test_spec, 1 subgoal):
 1. prog x t = ABR.insert x t
```

Un seul sous-but a été construit directement à partir de notre objectif. Isabelle est un prouveur, et ce sous-but est la formule à prouver. On ne peut pas prouver cette formule directement, mais on peut la tester. C'est ce que fait HOL-TestGen en appliquant une stratégie de sélection par découpage.

Cette méthode s'appelle `gen_test_cases` dans HOL-TestGen. Voila ce qu'elle renvoie :

```
   > apply(gen_test_cases 1 1 "prog")

   proof (prove) : step 1

   fixed variables : prog, x, t

   goal (test_spec, 9 subgoals) :

   1. prog ?X1X33 ET = MKT ?X1X33 ET ET
   2. THYP ((∃ x. prog x ET = MKT x ET ET) ⟶ (∀ x. prog x ET =
MKT x ET ET))
   3. [| ?X1X25 ≠ ?X3X27 ; ?X1X25 < ?X3X27 |] ⟶ prog ?X1X25 (MKT ?X3X27
ET ?X2X26) = MKT ?X3X27 (MKT ?X1X25 ET ET) ?X2X26
   4. THYP ((∃ x xa xb. x ≠ xb ⟶ x < xb ⟶ prog x (MKT xb ET
xa) = MKT xb (MKT x ET ET) xa) ⟶ (∀ x xa xb. x ≠ xb ⟶ x < xb
⟶ prog x (MKT xb ET xa) = MKT xb (MKT x ET ET) xa))
   5. prog ?X1X16 (MKT ?X1X16 ET ?X2X17) = MKT ?X1X16 ET ?X2X17
   6. THYP ((∃ x xa. prog x (MKT x ET xa) = MKT x ET xa) ⟶ (∀ x
xa. prog x (MKT x ET xa) = MKT x ET xa))
   7. [| ?X1X7 ≠ ?X3X9 ; ?X1X7 < ?X3X9 |] ⟶ prog ?X1X7 (MKT ?X3X9 ET ?X2X8)
= MKT ?X3X9 ET (ABR.insert ?X1X7 ?X2X8)
```

```
  8. THYP ((∃ x xa xb. x ≠ xb ⟶ ∕x < xb ⟶ prog x (MKT xb ET
xa) = MKT xb ET (ABR.insert x xa)) ⟶ (∀ x xa xb. x ≠ xb ⟶ ∕x
< xb ⟶ prog x (MKT xb ET xa) = MKT xb ET (ABR.insert x xa)))
9. THYP (Suc 0 < size t ⟶ prog x t = ABR.insert x t)
```

Expliquons un peu les arguments de `gen_test_cases` :

- Le premier argument est une profondeur de découpage. La procédure construit tous les cas de test possibles pour des entrées de test dont la taille est inférieure ou égale à cette valeur. Dans cet exemple, cet argument est à un, dont elle construit tous les cas possibles (en fonction des axiomes de la spécification) pour des arbres ne contenant au plus qu'un seul constructeur MKT.
- Le deuxième argument ne nous intéresse pas ici : il faut le laisser à 1.
- Le troisième argument est le nom de l'opération que l'on veut tester.

L'application de la stratégie `gen_test_cases` produit différents sous-buts. Certains correspondent aux tests abstraits (les sous-buts impairs). Ainsi le premier sous-but permet de tester l'insertion d'un élément dans l'arbre vide. Les autres buts sont les hypothèses de tests (les sous-buts pairs). Ces dernières servent à combler la distance entre la preuve de correction du programme et la vérification effectuée par les tests. Par exemple, le deuxième sous-but exprime que si le test d'insertion d'un élément dans l'arbre vide est en succès, le programme sera correct pour toute insertion d'un élément dans l'arbre vide. Cette hypothèse est issue du critère d'uniformité sur le premier sous-domaine (dans lequel l'arbre est vide). De même, la dernière hypothèse est issue du critère de régularité sur les arbres. Ces hypothèses permettent à Isabelle de disposer d'un ensemble de sous-buts équivalents au but de départ qui exprime la correction (complète) du programme.

Voici les quatre sous-buts générés qui nous intéressent. Pour plus de lisibilité, les variables de type `'a` sont notées à l'aide des lettres `x`, `y`, et les variables de type `'a tree` à l'aide de la variable `X`

```
  1. prog(x, ET) = MKT(x, ET, ET)
  3. [|x ≠ y ; x < y|] ⟹ prog (x, MKT( y, ET, X)) = MKT( y ,MKT
(x ,ET, ET), X)
  5. prog(x, MKT(x, ET, X) = MKT(x, ET, X)
  7. [|x ≠ y ; ¬ x < y|] ⟹ prog(x, MKT( y, ET, X)) = MKT(y, ET,
ABR.insert( x, X))
```

Rappelons que la notation `[|A1 ; ... ; An|] ⟹ B` est logiquement équivalente à $A_1 \wedge \cdots \wedge A_n \Rightarrow B$.

Il est bien sûr possible de générer des cas de tests pour des profondeurs plus grandes. Sur ce petit tableau les temps approximatifs pour générer les sous-buts

à une profondeur donnée dans l'exemple du test de la fonction `insert` dans les arbres binaires de recherche.

Profondeur	Temps nécessaire	Sous-buts
0	$< 1''$	3
1	$< 1''$	9
2	$2''$	45
3	$1'40''$	217
4	Des heures	1425

On s'aperçoit que le temps nécessaire à la génération des sous-buts pour une profondeur supérieure à 3 est beaucoup trop important. À chaque étape l'outil fait du dépliage sur toutes les sous branches possibles. Les auteurs de l'outil ont développé des techniques pour améliorer le temps de réponse en ajoutant des lemmes à la procédure de simplification d'Isabelle.

La commande `store_test_thm ‘‘test’’` permet de stocker les tests abstraits dans une variable $test$.

Puis la séquence suivante génère les données de tests pour les cas de tests contenus dans la variable $test$, et puis les affichent.

```
gen_test_data "test"
thm test.test_data
```

Voici les tests que l'on obtient comme tests concrets grâce à la deuxième ligne (c'est-à-dire le critère d'uniformité pour les sous-buts 1, 3, 5, et 7) :

```
prog 8 ET = MKT 8 ET ET
prog 4 (MKT 7 ET ET) = MKT 7 (MKT 4 ET ET) ET
prog 1 (MKT 1 ET (MKT 1 ET ET)) = MKT 1 ET (MKT 1 ET ET)
prog 9 (MKT 8 ET ET) = MKT 8 ET (MKT 9 ET ET)
```

Les valeurs assignées aux variables sont générées aléatoirement mais doivent respecter les pré-conditions. Comme pour les tests abstraits, cette phase peut donc durer très longtemps car il n'est pas forcément évident de trouver un test qui vérifie les pré-conditions. HOL-TestGen intègre des moyen de limiter le nombre d'itérations pour cette génération aléatoire. Il est ensuite possible de générer un script de test pour soumettre ces données de test à un programme `prog` (en SML).

HOL-TestGen est un outil de test par partition complet. Il utilise toute la puissance de Isabelle/HOL pour spécifier des opérations, permet de partitionner des ensembles de tests à partir de cette spécification, et intègre également des moyens de soumettre automatiquement les tests générés. Il a été utilisé dans de nombreux

cas de figures, et encore récemment dans [BW07]. Cependant, la prise en main de cet outil est un peu lourde, notamment à cause de son intégration dans Isabelle/HOL. La méthode de partition, si elle couvre bien les axiomes de la spécification, est plutôt lente dès qu'il s'agit de tester des données de grande taille. En effet, le découpage s'effectue dans toutes les directions en même temps, et le nombre de cas possibles explose très rapidement même pour des profondeurs peu conséquentes.

4.3 QuickCheck

QuickCheck est un outil développé par K. Claessen et John Hughes [CH00]. C'est un outil très léger qui fait environ 300 lignes de code. Il permet d'aider les programmeurs, dans le langage Haskell, à formuler et tester des propriétés (les objectifs de test) sur des programmes. Les propriétés sont décrites dans ce langage de programmation et peuvent être testées automatiquement de manière aléatoire. Contrairement à HOL-TestGen et à LOFT (voir section suivante), QuickCheck utilise une *méthode aléatoire* pour sélectionner les tests. Ainsi, afin de réaliser une bonne couverture des propriétés à vérifier, il faut que l'utilisateur sépare bien les composants à tester.

Pour décrire les propriétés, ils utilisent une spécification qui est intégrée dans le langage Haskell, et qui se trouve dans le même module que la fonction à tester. Les tests concrets sont générés aléatoirement. Les auteurs défendent la thèse selon laquelle les méthodes de test aléatoire sont finalement assez compétitives par rapport aux autres méthodes de tests. Pour améliorer les problèmes de distribution des valeurs que peuvent rencontrer les méthodes de test aléatoire, l'outil QuickCheck intègre des mécanismes de contrôle humains.

4.3.1 Un exemple simple

La syntaxe pour tester une fonction est assez simple. Si l'on veut tester l'opération classique *reverse* sur les listes, il suffit d'écrire la propriété que l'on veut tester dans le langage Haskell de la manière suivante :

```
prop_Rev x y =
  reverse (x ++ y) == reverse y ++ reverse x
```

avec ++ qui désigne la concaténation de deux listes.

Ensuite, l'outil s'occupe de tout, il suffit de taper la commande suivante :

```
quickCheck prop_Rev
```

L'outil génère par défaut 100 tests concrets qu'il soumet à l'implantation de l'opération *reverse*, et renvoie soit :

- `OK: passed 100 tests.`
  lorsque les 100 tests sont en succès,
- `Falsifiable`
  lorsque au moins l'un des tests est en échec. Il donne en plus les valeurs incriminées.

Il est également possible de tester des propriétés plus complexes que des équations, comme des formule conditionnelles de la forme $A ==> B$. QuickCheck génère alors 100 tests qui vérifient la condition A. Si il ne trouve pas 100 tests valides lors des 1000 premier essais, il renvoie simplement le nombre de tests trouvés.

4.3.2 Génération des données de tests

Pour les types de données classiques, Quickcheck intègre directement des mécanismes de génération de tests. Comme cette génération se fait de manière aléatoire, Quickcheck permet de classer les tests générés :

- détection des tests triviaux, par exemple les listes vides pour le type de données des listes,
- classement des tests en fonction de la taille des données, ce qui permet de voir si la génération aléatoire réalise une bonne couverture du domaine ou bien se restreint à des données de petite taille.

Pour éviter de générer, par exemple, des listes qui ont toujours une taille très petite et ainsi de passer à côté de plein d'erreurs potentielles, ou bien des listes trop grande et risquer de ne pas terminer, QuickCheck possède des mécanismes pour régler la fréquence d'apparition d'un élément et des bornes à ne pas dépasser.

En revanche, la génération des données de test est laissé aux utilisateurs pour des types définis par eux. Cela se fait à l'aide de l'opérateur Arbitrary, qui permet par exemple pour un type somme

```
data Couleur = Rouge | Bleu | Vert
```

de définir le générateur suivant :

```
instance Arbitrary Couleur where
 arbitrary = oneof
  [return Rouge, return Bleu, return Vert]
```

Les données de tests seront ainsi choisies parmi Rouge, Bleu et Vert.

4.3.3 Comparaison avec HOL-TestGen

L'intégration complète de QuickCheck dans le langage Haskell le rend très simple à utiliser et utilisable par tout programmeur. C'est son avantage principal face à HOL-TestGen qui est plus complexe. Rappelons que l'intégration de celui-ci dans Isabelle le rend assez lourd à utiliser. Par contre, il permet de réaliser une meilleure couverture des tests que QuickCheck puisqu'il realise un découpage précis à l'aide des axiomes, et il est plus facilement transposable dans des langages de programmation différents.

4.4 LOFT

Plus ancien que les outils précédents, LOFT [BGM91, Mar91b, Mar91a, Mar95] est un outil qui a été développé par Bruno Marre pour automatiser une méthode de sélection de tests par dépliage des axiomes (donc par *partition*). Il est développé dans le langage de programmation logique ECLIPSE [ECR06], qui est un langage de la même famille que PROLOG. L'idée était d'utiliser les principes de la programmation logique pour implémenter une méthode de sélection de tests à partir de spécifications algébriques.

Son noyau est une procédure de résolution équationnelle, qui permet de générer des tests à partir d'une formule équationnelle caractérisant un sous-domaine d'entrées d'une opération. Cette procédure est appelé surréduction conditionnelle ("narrowing" en anglais). Elle permet également de résoudre des problèmes équationnels dans une théorie décrite par des axiomes conditionnels positifs.

C'est cette approche qui a été à la base du travail que nous présentons dans le prochain chapitre de cette thèse. En effet, nos procédures de dépliage sont inspirées de ce que fait l'outil LOFT.

4.4.1 Les axiomes : définition et restrictions

Dans LOFT, à chaque axiome de la spécification est associé une clause de Horn [1] sans égalité. Reprenons l'exemple de la spécification des entiers naturels donnée section 1.4.1 page 36. Les axiomes définissant l'opération $+$ doivent tout d'abord être écrit de manière préfixe :

$$\begin{array}{rcl} add(0, x) & = & x \\ add(succ(x), y) & = & succ(add(x, y)) \end{array}$$

[1] En logique des prédicats, une clause de Horn est une formule de la forme $l_1 \wedge \cdots \wedge l_n \Rightarrow p$ avec l_i, p des formules atomiques. Ces clauses sont à la base des langages de programmation logique tels que PROLOG

Ensuite, pour être intégré dans LOFT, ils sont transformés en clauses de Horn sans prédicats d'égalité. La notation $: -$ est celle des langages de programmation logique tels que PROLOG. Elle remplace le symbole d'implication $\Leftarrow$ de la logique des prédicats du premier ordre ($a : - b$ se lit "a si b"). Les variables sont notées à l'aide des lettres majuscules.

$$add(0, M, M)$$
$$add(succ(N), M, succ(Z)) : - add(N, M, Z)$$

À toute opération définie d'arité n, LOFT associe une relation (un prédicat) d'arité $n + 1$, et à tout constructeur d'arité n un nom de fonction d'arité n. Toute les formules manipulées par LOFT devront être transformées de cette manière. On obtient ainsi un programme logique à partir d'une spécification conditionnelle positive.

La stratégie de LOFT permet de calculer, à partir de ces formules, des substitutions qui permettront de générer des tests. Cependant des solutions ne peuvent pas toujours être trouvées. Cette stratégie n'est valide que pour une classe particulière de spécifications conditionnelles positives. La signature $\Sigma = (S, F, V)$ est une signature avec un ensembles de constructeurs [2] C dans F. Les équations de la conclusion des axiomes sont de la forme $g(u_1, \ldots, u_n) = v$ telles que $g \in F \setminus C$, $u_i \in T_\Omega(V)$ avec $\Omega = (S, C)$, et toutes les opérations non-constructeurs dans v sont toutes plus petites que g selon un ordre bien-fondé $>_F$ construit sur F. La conclusion $g(u_1, \ldots, u_n) = v$ peut donc être orientée de gauche à droite (i.e. $g(u_1, \ldots, u_n) \rightarrow v$) comme une règle de réécriture. De plus, le système de réécriture conditionnelle associé à ces axiomes forme un système qui termine et qui est confluent. Pour simplifier, les axiomes doivent être "exécutables" pour pouvoir être compris par LOFT (nous étudierons en détail les restrictions dans le dernier chapitre de cette thèse page 139).

4.4.2 Le dépliage

Ce qui nous intéresse dans cet outil, c'est sa méthode de dépliage. Elle utilise la structure des axiomes de la spécification, et un but qui est une formule qui caractérise un objectif de test. Nous allons la présenter succinctement ici dans un cadre restreint, celui de l'outil LOFT. Nous l'étendrons dans le chapitre 6.

Exemple 4.4.1 *Considérons la spécification de l'addition donnée précédemment à l'aide des deux clauses de Horn. LOFT permet de déplier un but construit à l'aide de l'opération* add*. Par exemple, l'équation* $add(n, n) = r$*, avec* n, m, r *des variables, peut être utilisée comme un objectif de test (un but dans LOFT). Cette*

[2] Les constructeurs sont des symboles d'opérations qui ne sont pas définis par les axiomes

équation caractérise tous les tests de cette forme (obtenus par substitution des variables et conséquences des axiomes). Le but $add(N, M, R)$ *est alors soumis à LOFT. Le dépliage de l'opération* add*, dans le but* $add(N, M, R)$*, permet de construire les deux sous-buts suivants :*

- *Le premier correspond au premier axiome de l'opération* add*. La substitution des variables est donnée par les unificateurs de* $N = 0$ *et* $R = M$ *et le sous-but est vide. Les tests ainsi décrits sont donc toutes les équations de la forme* $add(0, r) = r$ *où* r *est substituée par un terme construit sur les constructeurs de entier (*0 *et* $succ$*).*
- *Le deuxième correspond au deuxième axiome. La substitution des variables est donnée par les unificateurs de* $N = succ(N')$*,* $R = succ(R')$ *où* N' *et* R' *sont des nouvelles variables. Le nouveau sous-but est* $add(N', M, R')$*. Les tests ainsi décrits sont de la forme* $add(succ(n'), m) = succ(r')$*, où* n', m *et* r' *sont substituées par des termes construits sur* 0 *et* $succ$ *et* $add(n', m) = r$ *est une conséquence des axiomes de l'addition.*

Comme le premier sous-but est vide, il ne peut plus être déplié. Au contraire, le deuxième sous-but construit par LOFT peut être déplié de nouveau selon les axiomes de l'addition :

- *Un premier sous-but vide est obtenu avec la substitution des variables construite selon l'unification de* $N = succ(0)$ *et* $R = succ(M)$*. Il caractérise les tests de la forme* $add(succ(0), m) = succ(m)$ *où* m *est substituée par tout terme construit sur* 0 *et* $succ$*. Ce sous-but étant vide, il ne pourra plus être déplié.*
- *Un deuxième sous-but est obtenu avec la substitution des variables obtenue par l'unification de* $N = succ(succ(N''))$ *et* $R = succ(succ(R''))$*. Le nouveau sous-but est* $add(N'', M, R'')$*. Il caractérise l'ensemble des tests de la forme* $add(succ(succ(n)), m) = succ(succ(r))$ *où* n, m *et* r *sont substituées par des termes construit sur* 0 *et* $succ$*. Comme dans la première étape de dépliage, ce cas pourrait être dépliée à nouveau sur les axiomes de l'addition.*

Ainsi, en stoppant le processus après deux dépliages successifs de l'addition, on obtient les 3 sous-buts ci-dessus construits à partir du but de départ.

Dans LOFT, l'opération de dépliage est réalisée à l'aide de la commande suivante :

```
unfold_std([#('f: s1, ..., sn -> s',m), ... ], eq )
```

avec f un nom d'opération muni de son profil $s_1 \times \ldots s_n \rightarrow s$, qui est l'opération à déplier, eq qui est une équation qui représente un but (l'objectif de test), et m la profondeur de dépliage souhaitée.

Exemple 4.4.2 *Dans l'exemple de l'opération d'addition, l'application de la fonction de dépliage (sur deux étapes) va se faire de la manière suivante :*

```
 ??- unfold_std([#('add: nat, nat -> nat',2)], add(N,M) = R).

FINAL BINDING:
N:nat = 0
R:nat = M:nat
CPUTIME = 0
;

FINAL BINDING:
N:nat = succ(0)
R:nat = succ(M:nat)
CPUTIME = 0
;

FINAL BINDING:
N:nat = succ(succ(_v0:nat))
R:nat = succ(succ(_v1:nat))

REMAINING CONSTRAINTS = {
   add(_v0:nat, M:nat) = _v1:nat
}
CPUTIME = 0
;

GLOBAL TIME ELAPSED = 17
NUMBER OF SOLUTIONS = 3
yes
```

Les trois `FINAL BINDING` *trouvés correspondent bien aux trois sous-buts obtenus par deux étapes de dépliage.*

Dans cet exemple simple, le dépliage n'est effectué que sur l'opération add mais dans d'autres cas il est possible de déplier n'importe quelle opération définie et permet donc à l'utilisateur de guider la sélection par dépliage. Par exemple, pour la définition de l'opération $*$ à l'aide de l'axiome $x * succ(y) = (x * y) + x$, la partie droite utilise l'opération $+$. Après une étape de dépliage, on pourra donc déplier soit cette opération soit l'opération initiale.

La solution donnée par LOFT après une ou plusieurs étapes de dépliage n'est pas nécessairement close. Il reste des variables qui ne sont pas instanciées. L'instanciation complète des variables peut être forcée et on obtient ainsi, non plus des sous-buts non résolus, mais bien des valeurs qui pourront être soumises à un programme.

Exemple 4.4.3 *Voici la commande à taper pour obtenir le résultat voulu pour le dépliage de l'opération d'addition :*

```
 ??- unfold_std([#('add:nat,nat -> nat',2)], add(N,M) = R), ?,
?(is_a_nat(M) = true).

FINAL BINDING:
N:nat = 0
M:nat = succ(succ(succ(0)))
R:nat = succ(succ(succ(0)))
CPUTIME = 17
;

FINAL BINDING:
N:nat = succ(0)
M:nat = succ(succ(succ(succ(0))))
R:nat = succ(succ(succ(succ(succ(0)))))
CPUTIME = 0
;

FINAL BINDING:
N:nat = succ(succ(0))
M:nat = succ(0)
R:nat = succ(succ(succ(0)))
CPUTIME = 16
;

GLOBAL TIME ELAPSED = 50
NUMBER OF SOLUTIONS = 3
yes
```

Les tests obtenus sont donc au nombre de trois, un par sous-but construit :
$add(0, succ(succ(succ(0)))) = succ(succ(succ(0)))$
$add(succ(0), succ(succ(succ(succ(0))))) = succ(succ(succ(succ(succ(0)))))$
$add(succ(succ(0)), succ(0)) = succ(succ(succ(0)))$

L'outil LOFT, plus ancien que HOL-TestGen et QuickCheck, permet plus que ces deux autres outils de s'intéresser à la sélection par dépliage. En effet, l'utilisateur peut déplier où il le souhaite, et il est donc sur ce point plus précis que HOL-TestGen.

C'est donc la finesse de cette méthode de dépliage de l'outil LOFT qui nous a le plus intéressée. Cependant, la classe de spécifications utilisée par LOFT doit respecter des conditions très restrictives sur la forme des axiomes. Dans les prochains chapitres, nous verrons comment la procédure de dépliage peut être étendue à des spécifications beaucoup moins restrictives.

La procédure de sélection de LOFT a été récemment réutilisée dans l'outil GATeL [MA00, MB04] qui permet de produire des tests à partir de spécifications LUSTRE. LUSTRE est un langage largement utilisé dans l'industrie pour construire des systèmes réactifs. L'outil GATeL déplie les équations LUSTRE pour obtenir des sous-domaines de test également caractérisé par des contraintes. Les contraintes sont résolus en générant aléatoirement un test pour chaque sous-domaine.

Chapitre 5

Notre approche du test

Dans ce chapitre, nous allons présenter plus particulièrement notre approche pour tester un programme à partir de sa spécification formelle. Comme nous l'avons vu dans le chapitre précédent, la sélection des tests est une nécessité dans le cadre du test. Afin de garantir la qualité de nos méthodes de sélection, nous allons présenter un cadre formel général pour le test à partir de spécifications axiomatiques. Nous nous plaçons dans le cas particulier des spécifications algébriques (vue dans la section 1.4 page 35).

Les spécifications algébriques fournissent un langage non ambigu qui est une référence solide pour garantir la correction d'un programme. Cette affirmation repose sur l'idée simple que les **modèles** de la spécification représentent des **programmes** acceptables. Cependant, nous ne pouvons pas comparer directement le programme et sa spécification. En effet, le programme est souvent plus riche en détails de fonctionnement que la spécification qui ne donne pas forcément toutes les informations nécessaires pour être directement exécutée sur une machine. Cette distance est embarrassante dans le cadre du test puisque l'idée intuitive que nous venons de donner est que le programme est représenté par un modèle de la spécification. Nous devrons donc nous accorder sur des moyens de comparer programme et spécification. L'hypothèse considérée pour tester un programme est de considérer que le programme doit avoir la même interface que la spécification (c'est-à-dire la même signature), et être déterministe (c'est-à-dire qu'un même calcul renvoie toujours la même valeur). De plus, pour être comparable avec la spécification le langage de programmation doit fournir un prédicat de comparaison entre deux valeurs de même type. Ce prédicat d'égalité est dit **observable** par la spécification ; c'est la partie du programme où le comportement des deux entités est censé être le même. C'est sur cette partie commune que nous comparerons le comportement du programme avec celui attendu par la spécification. Les tests seront donc des formules observables validées par la spécification. Cette partie observable du programme constituera la base pour la définition d'un

jeu de tests de référence à partir duquel nous pourrons commencer un processus de sélection de tests.

La première section de ce chapitre est dédiée à la formalisation de notre approche du test, et la deuxième aborde plus en détail le problème des critères de sélection d'un jeu de tests.

5.1 Test à partir de spécifications algébriques

Dans cette section, nous allons décrire un cadre formel de test précédemment étudié dans [LGA96, ALG02, BGM91, Arn97] et montrer comment construire nos tests à partir d'une spécification algébrique.

5.1.1 Rappel : les spécifications algébriques

Les spécifications algébriques étudiées dans la section 1.4 page 35 permettent de décrire le comportement d'un programme à l'aide d'axiomes.

Exemple 5.1.1 *Plaçons nous dans le cadre de la spécification $Piles$ que nous avons donnée section 1.4.3 page 38. Nous voulons être capable, par exemple, de vérifier qu'un programme, où est implanté l'opération $hauteur$ d'une pile, a bien un comportement correct vis à vis des axiomes de la spécification $Piles$. Par exemple, nous voudrions être sûr que l'opération $hauteur$ du programme appliquée sur une pile vide rend bien 0, c'est-à-dire que l'axiome $hauteur(vide) = 0$ est bien vérifié.*

5.1.2 Programmes

La soumission d'un test sur un programme doit permettre de vérifier la correction locale du programme. Un test soumis à un programme doit pouvoir être exécuté. Le résultat obtenu doit alors pouvoir être comparé au résultat attendu (défini par la spécification) et pour cela disposer d'un moyen pour pouvoir trancher quant au succès ou à l'échec d'un test. C'est ce que l'on appelle le *problème de l'oracle*.

Pour permettre la comparaison entre un programme P et une spécification $SP = (\Sigma, Ax)$, nous interprétons le programme comme un Σ-modèle (voir section 1.1.2 page 19). Ceci n'a de sens que si le programme a un comportement fonctionnel sur la signature Σ, c'est-à-dire si toute formule de $For(\Sigma)$ peut être calculée par le programme de manière déterministe.

Par la suite, nous assimilerons un programme P et son interprétation comme un Σ-modèle ($P \in Mod(\Sigma)$. Le soumission et le résultat (succès ou échec) d'un

test $t \in For(\Sigma)$ sur un programme P est donc assimilée à la validation $P \models t$. C'est l'un des avantages du test à partir de spécifications car il fournit à la fois la donnée de test à fournir au programme et le résultat attendu par la spécification.

Exemple 5.1.2 *Imaginons que nous voulions tester un programme P de la fonction $hauteur$ pour les $Piles$. Comme P est un Σ-modèle, il est possible par exemple de vérifier que l'axiome $hauteur(vide) = 0$ est bien validé par le programme, puisque c'est une formule logique construite sur la signature Σ. Nous avons donc un moyen de dire si $P \models hauteur(vide) = 0$. En pratique, cela consiste à soumettre au programme successivement $hauteur(vide)$ et 0 puis de comparer les deux résultats obtenus. Cela n'est possible que si il existe un prédicat $=$ prédéfini du langage pour comparer des données de ce type, c'est-à-dire, dans ce cas, où la sorte des entiers est observable.*

5.1.3 Observabilité

Le programme et la spécification partagent la même signature, et les tests sont des formules construites sur cette signature. Cependant nous devons être capable de dire si une formule qui représente un test est vraie ou fausse pour le programme. Or le programme ne permet pas forcément de rendre compte du comportement de toutes les opérations et prédicats de la spécification.

Du point de vue du test fonctionnel, seul le comportement observable du programme est intéressant. Nous considérerons donc une classe de formules que le programme peut manipuler : les formules *observables* (ou exécutables). En pratique, les formules observables sont des formules closes qui ne font intervenir qu'un ensemble donné de sortes dites observables (par exemple, toutes les équations pour lesquels le prédicat d'égalité est inclus dans le langage de programmation).

Nous disposons donc d'un ensemble $Obs \subseteq For(\Sigma)$ de formules dites observables. Ce sont les formules que l'on peut tester sur le programme.

Exemple 5.1.3
Dans le cadre des spécifications conditionnelles positives, nous pouvons supposer que nos tests sont simplement les équations closes entre sortes observables conséquences de la spécification, c'est-à-dire $Obs = \{t = t' \mid t, t' \in T_\Sigma\}$ avec t, t' des termes de sorte observable.

Ainsi, pour la spécification des piles, nous considérons que les équations closes $hauteur(vide) = 0$ ou $depiler(empiler(3, vide)) = vide$ sont des formules observables.

Il est également possible de tester des formules qui contiennent des données de sortes non observables. Cela se fait à l'aide de contexte observable, c'est-à-dire d'un contexte $c[.]_\omega$ (voir définition 2.1.2 page 42) de sorte s observable pour

laquelle on dispose du prédicat d'égalité. Le terme placé à la position ω est de sorte non observable.

Exemple 5.1.4 *Dans l'exemple des piles, supposons que les entiers soient observables par le langage de programmation, mais pas les piles. Le contexte observable construit à partir* $hauteur$ *permet de tester les piles via les entiers ; par exemple pour l'axiome* $depiler(vide) = vide$*, que l'on peut tester à l'aide de l'égalité suivante* $hauteur(depiler(vide)) = hauteur(vide)$ *car les entiers sont observables.*

Dans la suite, nous n'entrerons pas dans les détails d'une telle approche et travaillerons sur des critères de sélection où toutes les sortes mises en présence sont observables.

5.1.4 Forme de nos tests

La soumission d'un test sur un programme doit comprendre un verdict qui nous permet de décider si le test est en succès ou en échec. Les tests pour une spécification $SP = (\Sigma, Ax)$ sont des formules construites sur la signature Σ. La formes des tests dépend donc du langage utilisé.

Dans le cas particulier de spécifications construites à partir d'axiomes équationnels ou conditionnels positifs, nos tests sont des équations entre termes clos. Cela pourrait être des formules conditionnelles. On se restreint aux équations à des fins de facilité de soumission.

Exemple 5.1.5 *Soit* $Piles = (\Sigma_{Piles}, Ax_{Piles})$ *la spécification des piles que l'on vient de définir, et soit* P_{Piles} *un programme qui réalise les mêmes opérations que la spécification sur les piles (empilement, dépilement, hauteur, etc ...). Pour la spécification* $Piles$*,* $hauteur(vide) = 0$ *ou* $depiler(empiler(3, vide)) = vide$ *sont des tests, mais* $hauteur(x) = n$ *n'en est pas un car il contient des variables qui ne sont pas observables par le programme.*

Les équations que nous allons voir comme des tests sont construites à partir d'une opération définie de la spécification. En opposition aux opérations constructeurs qui ne sont pas définies à partir d'axiomes mais permettent de construire les structures de données. Les tests que nous allons privilégier seront donc de la forme $f(u_1, \ldots, u_n) = v$ avec f une opération définie, $u_1, \ldots, u_n, v \in T_\Sigma$, avec $u_1, \ldots, u_n$ les données que nous allons soumettre au programme et v le résultat attendu par la spécification.

5.1.5 Correction d'un programme par rapport à sa spécification

Tester un programme, c'est essayer de trouver des erreurs dans celui-ci. Tester un programme à partir de sa spécification, c'est essayer de montrer qu'il est incorrect par rapport à elle. La spécification permet de déduire des formules (des tests) qui si elles sont validées par le programme nous renseignent sur la correction de celui-ci. Si toutes les formules que l'on peut déduire de la spécification (à observation près), sont des conséquences du programme (un Σ-modèle), alors nous allons pouvoir assurer correction (partielle ou totale) pour ce programme.

Cette notion de correction d'un programme par rapport à sa spécification peut être formalisée. Pour être dit correct par rapport à une spécification, un programme doit être observationnellement équivalent à un modèle de la spécification pour les formules observables de Obs.

Définition 5.1.1 (Correction) *Soit $SP = (\Sigma, Ax)$ une spécification, P un programme qui est un Σ-modèle, et Obs un ensemble de Σ-formules. P est* **correct** *pour SP via Obs, noté $Correct_{Obs}(P, SP)$, si et seulement si il existe un modèle $\mathcal{M} \in Mod(SP)$ telle que $\mathcal{M} \equiv_{Obs} P$. Ce qui signifie que les deux modèles sont équivalents*[1] *pour l'ensemble de formules Obs.*

5.1.6 Les tests

Comme nous l'avons vu dans les sections 5.1.3 et 5.1.4, les jeux de tests sont composés de formules observables ayant une forme particulière. Pour être utilisables, ces jeu de tests doivent être en succès sur un programme P correct, c'est-à-dire valides pour le Σ-modèle associé à P. On parle alors de jeux de tests **non-biaisés**. Nous allons pour cela considérer que les tests sont des conséquences de la spécification. Nous pouvons les définir de manière formelle :

Définition 5.1.2 *Un* **test** *est une formule de $SP^{\bullet} \cap Obs$, c'est-à-dire une formule observable conséquence sémantique de la spécification.*

Un **jeu de tests** *est un ensemble de tests. Un jeu de tests T est en* **succès** *sur un programme P si et seulement si $\forall \varphi \in T, P \models \varphi$, sinon on dit qu'il est en* **échec**. *On notera $P \models T$ lorsque T est en succès sur P.*

Exemple 5.1.6
$hauteur(vide) = 0$ ou $depiler(empiler(3, vide)) = vide$ sont des tests pour la spécification SP_{Piles}, mais $depiler(empiler(3, vide)) = empiler(3, vide)$ n'en

[1] La notion d'équivalence $\equiv_{\Psi}$ entre deux modèles est donnée dans la définition 1.1.10 page 21

est pas un, car la formule n'est pas une conséquence sémantique de la spécification. Cela signifie qu'elle ne correspond pas à un comportement valide pour la spécification. Si nous mettions un tel test en oeuvre sur un programme, son échec ne serait pas synonyme d'erreur. Il détecterait une erreur qui n'existe pas si le programme est correct, et pourrait ne pas détecter une erreur sur un programme qui en contient une.

Un jeu de test est dit **non-biaisé** si et seulement si ce jeu de tests est en succès sur tout programme. C'est nécessairement le cas pour les tests de $SP^\bullet \cap Obs$.

Proposition 5.1.1 *Soient $SP = (\Sigma, Ax)$ une spécification algébrique, Obs un ensemble de formules de $For(\Sigma)$, P un programme vu comme un Σ-modèle et T un jeu de test de $SP^\bullet \cap Obs$.*

Pour tout programme P observable via Obs, si P est correct par rapport à SP via Obs alors T est en succès sur P.

$$Correct_{Obs}(P, SP) \Rightarrow P \models T$$

Preuve

Par définition, P est un Σ-modèle de $Mod(\Sigma)$ donc il existe un modèle $M \in Mod(SP)$ tel que $M \equiv_{Obs} P$. Nous savons également que $T \subset SP^\bullet \cap Obs$ et que toutes les formules de $SP^\bullet \cap Obs$ sont valides dans $Mod(SP)$. Donc toutes les formules de T sont valides dans le modèle M donc dans P, donc par définition $P \models T$.

★

$SP^\bullet \cap Obs$ est le plus grand ensemble de formules qui est à la fois satisfait par tous les SP-modèles, et exécutable par n'importe quel programme sous test capable d'interpréter les formules de Obs.

5.1.7 Comparaison des jeu de tests

Le but du test étant de détecter des erreurs, il est intéressant de comparer les différents jeux de tests en fonction des propriétés qu'il permettent de vérifier. Nous pouvons comparer leur efficacité, laquelle considère qu'un jeu de tests T est plus efficace qu'un jeu de tests T' si son succès sur un programme implique que le jeu de tests T' soit également en succès.

Définition 5.1.3 *Soient SP une spécification, Obs un ensemble de formules observables, et T, T' deux jeux de tests.*

T est **plus efficace** *que T' si et seulement pour tout programme $P \in Mod(\Sigma)$ on a $P \models T \Rightarrow P \models T'$. On note $T \leq T'$.*

T et T' sont **équivalents** *si et seulement si $T \leq T'$ et $T' \leq T$. On notera $T \approx T'$.*

5.1.8 Exhaustivité d'un jeu de tests

Quand un jeu de tests en succès sur un programme permet de garantir la correction de celui-ci, on dit que ce jeu de tests est exhaustif.

Définition 5.1.4 (Exhaustivité) *Un jeu de tests T est* **exhaustif** *pour $SP = (\Sigma, Ax)$ via Obs si et seulement si*

$$\forall P \in Mod(\Sigma), P \models T \Leftrightarrow Correct_{Obs}(P, SP)$$

Il n'existe pas nécessairement un jeu de tests exhaustif. Cela dépend de la nature de la spécification SP et de l'ensemble Obs. De plus, un tel jeu de tests est souvent infini. Son existence signifie simplement que la correction d'un programme peut être approchée asymptotiquement. Son intérêt est que n'importe quelle erreur d'un programme est susceptible d'être découverte. Il permet également de démarrer un processus de sélection d'un jeu de tests de taille fini comme nous le verrons dans la section 5.2.3 de ce chapitre, puis dans le chapitre 6.

5.2 Les critères de sélection

L'idée de définir des critères de sélection vient du problème de la taille des jeux de tests que l'on doit gérer. Comme il n'est pas possible de tout tester, il faut faire des choix, et donc appliquer des critères les plus judicieux possibles. Dans notre cas, nous appliquons des *critères de sélection* sur un jeu de test de référence dans le but d'extraire un jeu de tests de taille raisonnable pour être soumis à un programme. L'idée sous-jacente est que tous les tests qui satisfont un critère de sélection révèlent la même classe d'erreurs.

Nous allons tout d'abord présenter les critères d'uniformité et de régularité [BGM91, BCFG86] qui sont des critères très simples à mettre en oeuvre. Ensuite nous formaliserons cette notion de critère de sélection, ce qui nous permettra d'y inclure des critères plus complexes et surtout de pouvoir garantir certaines propriétés sur ces critères.

5.2.1 Critère d'uniformité

Comme nous l'avons dit précédemment, en général nous manipulons des jeux de tests abstraits construits à partir d'un objectif de test. Cet objectif est une formule de $For(\Sigma)$ (souvent un axiome). Or les formules (c'est-à-dire les tests) que l'on peut soumettre à un programme sont des formules sans variables.

Le *critère d'uniformité* s'appuie sur l'idée qu'un ou plusieurs tests, choisis de manière arbitraire pour un objectif de test permettent de mettre en évidence une

même classe d'erreur. La validation de ces tests pour le programme vaut pour la validation de l'objectif de test. C'est ce que l'on appelle l'hypothèse d'uniformité. Formalisons cette notion :

Définition 5.2.1 *Soit φ une formule de $For(\Sigma)$. Un jeu de tests T vérifie le* **critère d'uniformité** *sur φ si et seulement si*

$$\{\sigma_i(\varphi) \mid \forall i, 1 \leq i \leq k, \exists \sigma_i : V \rightarrow T_\Sigma, SP \models \sigma_i(\varphi)\} \in T$$

*avec k le nombre de tests à choisir pour la formule φ. L'ensemble des jeux de tests satisfaisant ce critère est noté $C_{U,k}(\varphi)$. Il est possible de formuler l'***hypothèse d'uniformité** *suivante :*

$$(T \in C_{U,k}(\varphi) \wedge P \models T) \Rightarrow (\forall \sigma : V \rightarrow T_\Sigma, SP \models \sigma(\varphi) \Rightarrow P \models \sigma(\varphi))$$

Exemple 5.2.1 *Plaçons nous encore une fois dans le cas de la spécification des piles (voir section 1.4.3). Soit la formule $hauteur(x) = y$ avec x, y des variables. Le jeu de test*

$$T = \left\{ \begin{array}{l} hauteur(vide) = 0, hauteur(empiler(succ(0), vide)) = 1, \\ hauteur(depiler(empiler(succ(succ(0)), vide))) = 0 \end{array} \right\}$$

satisfait le critère d'uniformité $C_{U,3}(hauteur(x) = y)$. Dans ce cas, la variable k vaut 3, il faut donc trois tests pour qu'un jeu de tests satisfasse le critère. Par hypothèse d'uniformité, il est possible d'affirmer que si un programme P valide le jeu de tests T alors il valide toutes les formules closes conséquences de la spécification de la forme $hauteur(x) = y$ (c'est-à-dire qu'il y a une substitution qui transforme les deux variables x, y par des termes clos).

Il existe d'autres critères dont le fonctionnement est similaire à celui du critère d'uniformité. On parle par exemple du *critère d'uniformité sur les constructeurs*. Sa seule différence est de considérer que les substitutions σ_i à construire doivent l'être vers des termes sur les constructeurs uniquement.

5.2.2 Critère de régularité

Le *critère de régularité* joue sur la taille des termes, et permet de construire tous les tests dont la taille des termes est inférieure à une valeur donnée. Pour cela, on introduit une notion particulière de taille.

Définition 5.2.2 *Soit t un terme de $T_\Sigma(V)$. La* **taille***, pour une sorte s donnée, d'un terme t est le nombre d'opérations $f : s_1 \times \cdots \times s_n \rightarrow s$ (à co-domaine dans s) apparaissant dans t. On la notera $|t|_s$.*

Exemple 5.2.2

$|hauteur(vide)|_{pile} = 1$ *car seule l'opération* $vide$ *est à co-domaine dans* $pile$*,*

$|hauteur(depiler(empiler(succ(succ(0)), vide)))|_{pile} = 3$ *car* $depiler$ *et* $empiler$ *sont à co-domaine dans* $pile$*,*

$|hauteur(empiler(succ(0), vide))|_{entier} = 2$ *car* 0 *et* $succ$ *sont à co-domaine dans* $entier$*.*

Le critère de régularité d'ordre k pour une sorte s construit pour toute variable de sorte s, apparaissant dans un objectif de test φ, tous les termes de taille inférieure ou égale à trois. Puis ensuite, il génère un test abstrait pour toutes les combinaisons de termes possibles, et les instancient à l'aide du critère d'uniformité.

Définition 5.2.3 *Soit φ une formule de $For(\Sigma)$. Un jeu de tests T vérifie le* **critère de régularité d'ordre k pour une sorte s** *sur φ si et seulement si*

$$\forall \psi \in Reg_{k,s}(\varphi), (T \in C_{U,1}(\psi))$$

tel que $Reg_{k,s}(\varphi) = \{\sigma(\varphi) \mid \sigma \in Sub_{k,s}\}$
avec $Sub_{k,s} = \{\sigma : V_s \rightarrow T_\Sigma(V) \mid \forall x \in V_s, |\sigma(x)| \leq k\}$.

On notera $C_{R,s,k}(\varphi)$ l'ensemble des jeux de tests satisfaisants ce critère de régularité.

$Sub_{k,s}$ est l'ensemble des substitutions $V_s \rightarrow T_\Sigma(V)$ qui associent à toute variable x de sorte s un terme de taille au plus k, ne comportant que des opérations à co-domaine dans s et des variables distinctes pour tout sous-terme de sorte autre que s. $Reg_{k,s}(\varphi)$ construit tous les termes possibles à partir de toutes les substitutions $Sub_{k,s}$ de taille inférieure à k pour une formule φ donnée.

Exemple 5.2.3 *Soit $hauteur(x) = y$ notre objectif de test. Pour construire le critère de régularité d'ordre 3 (par exemple) pour la sorte pile, on considère d'abord l'ensemble $Sub_{3,pile}$ qui contient pour toutes les variables de sorte s l'ensemble des substitutions vers des termes de taille inférieure à 3. En considérant toutes les opérations dont le co-domaine est celui des piles, on obtient les termes suivants :*

$vide$
$empiler(a, vide)$

$depiler(vide)$
$empiler(a, empiler(b, vide))$
$empiler(a, depiler(vide))$
$depiler(empiler(a, vide))$

avec a, b *des variables de type* $entier$*. On obtient donc l'ensemble de tests abstraits suivant :*

$$\begin{aligned} Reg_{3,pile}(hauteur(x) = y) \quad = \{ \quad & hauteur(vide) = y \\ & hauteur(empiler(a, vide)) = y \\ & hauteur(depiler(vide)) = y \\ & hauteur(empiler(a, empiler(b, vide))) = y \\ & hauteur(empiler(a, depiler(vide))) = y \\ & hauteur(depiler(empiler(a, vide))) = y \ \} \end{aligned}$$

Il ne reste plus qu'à substituer à toutes les variables un terme clos en utilisant le critère d'uniformité vu précédemment. On obtient alors le jeu de tests T *satisfaisant le critère* $C_{R,3,pile}(hauteur(x) = y)$ *suivant :*

$$\begin{aligned} T \quad = \{ \quad & hauteur(vide) = 0 \\ & hauteur(empiler(succ(0), vide)) = succ(0) \\ & hauteur(depiler(vide)) = 0 \\ & hauteur(empiler(succ(succ(0)), empiler(0, vide))) = succ(succ(0)) \\ & hauteur(empiler(succ(0), depiler(vide))) = succ(0) \\ & hauteur(depiler(empiler(succ(succ(0)), vide))) = 0 \ \} \end{aligned}$$

Le choix des valeurs pour les variables est donné par le critère d'uniformité.

Ce critère peut également être restreint à une version où l'on ne considère que les termes construits à l'aide des constructeurs d'une sorte donnée. On parle alors de *critère de régularité sur les constructeurs*.

5.2.3 Formalisation

Comme nous l'avons vu dans la section précédente, les jeux de tests sont construits à l'aide d'une propriété φ qui est un axiome, ou n'importe quelle formule choisie comme *objectif de test*. Tous les tests qui peuvent être construits à partir d'un objectif de test forment un ensemble qui caractérise un jeu de tests. Celui-ci contient, pour un objectif donné φ, tous les tests $\sigma(\varphi)$ tels que $\sigma(\varphi) \in SP^{\bullet} \cap Obs$.

Ce jeu de tests peut être de taille infinie. Il est possible de le couvrir partiellement par des tests sélectionnés à l'aide des critères d'uniformité et de régularité, mais ce sont des méthodes où l'aléatoire joue un grand rôle.

Les méthodes de sélection que nous utilisons dans cette thèse sont des méthodes par découpage d'un jeu de tests en une famille de sous-jeux de tests. Dans ce cadre, un critère de sélection est en fait une application qui effectue ce découpage. Il nous faut des outils pour raisonner sur ces critères de sélection.

Le critère de sélection C est une application qui découpe un jeu des tests de départ T en une famille de sous-jeux de tests $\{T_i\}_{i \in I_{C(T)}}$ avec $I_{C(T)}$ un ensemble d"'indices" résultant de l'application du critère de sélection C sur le jeu de tests T.

Définition 5.2.4 (Critère de sélection) *Un* **critère de sélection** *est une application $C : \mathcal{P}(SP^{\bullet} \cap Obs) \rightarrow \mathcal{P}(\mathcal{P}((SP^{\bullet} \cap Obs))$ [2]. Pour un jeu de tests T on note $C(T) = \{T_i\}_{i \in I_{C(T)}}$. On note alors $|C(T)| = \cup_{i \in I_{C(T)}}(T_i)$ l'ensemble des tests obtenus après l'application du critère de sélection C sur T.*

Le nombre de sous-jeux de tests T_i construits par découpage dépend du critère de sélection et du jeu de tests de départ. Les jeux de tests T_i caractérisent des sous propriétés de T, c'est-à-dire des sortes de sous objectifs de tests. Comme pour le jeu de tests de départ ils peuvent être couverts par les critères d'uniformité et de régularité. Cela consistera à sélectionner un ou plusieurs tests pour chaque sous-jeu de tests construit.

Exemple 5.2.4 *Soit φ un objectif de test. Le critère de régularité $C_{R,s,k}(\varphi)$, pour une taille k et une sorte s données, nous donne un ensemble de tests abstraits $Reg_{k,s}(\varphi)$ qui caractérise tous les tests pour des données de taille inférieure à k. Il est possible, par exemple, de définir un critère de sélection C_1 qui découpe le jeu de tests abstrait $Reg_{k,s}(\varphi)$ en n sous-jeux de tests $Reg_{i,s}(\varphi)$ pour $1 \leq i \leq k$. Un jeu de tests T vérifie le critère C_1 si et seulement si*

$$\bigcup_{1 \leq i \leq k} \{C_{U,1}(\psi) | \forall \psi \in Reg_{i,s}(\varphi)\}$$

5.2.4 Propriétés

La définition générale que nous venons de donner permet de décrire une large classe de critères de sélection par découpage. Afin de garantir l'efficacité d'un critère particulier, il est possible de vérifier des propriétés sur celui-ci :

Définition 5.2.5 *Soit C, C' des critères de sélection, et T, T' des jeux de tests :*

- *C est* **correct** *pour T si et seulement si $|C(T)| \subseteq T$,*
- *C est* **complet** *pour T si et seulement si $|C(T)| = T$,*

[2] Soit un ensemble E, $\mathcal{P}(E)$ désigne l'ensemble des tous les sous-ensembles de E

– C **partitionne** T *si et seulement si* $\forall i, j \in I_{C(T)}, i \neq j \Rightarrow T_i \cap T_j = \emptyset$,

Les propriétés de correction et complétude sont essentielles pour qu'un critère de sélection soit utilisable : la correction assure que les tests sont bien sélectionnés dans le jeu de tests de départ (aucun test n'est ajouté), et la complétude assure que nous conservons bien tous les tests (aucun test n'est perdu). Quand on partitionne T avec C, cela signifie que les T_i ne se superposent pas, et assure ainsi que l'on ne sélectionnera pas deux tests identiques.

Exemple 5.2.5 *Le critère C_1 de l'exemple 5.2.4 de la section précédente est un critère de sélection correct et complet au sens de la définition que l'on vient de donner. L'ensemble des tests caractérisé par le découpage est exactement le même que celui du jeu de tests de départ. Cependant, ce critère C_1 n'est pas un critère qui partitionne. Les sous-jeux de tests construits ne sont pas distincts les uns des autres.*

Pour obtenir un critère qui partitionne vraiment, on pourrait par exemple construire un critère de régularité d'ordre k identique à celui de la définition 5.2.3 mais où l'on remplace $Sub_{k,s}$ par $Sub'_{k,s}$ telle que

$$Sub'_{k,s} = \{\sigma : V_s \rightarrow T_\Sigma(V) \mid \forall x \in V_s, |\sigma(x)| = k\}$$

Ce critère $C'_{R,s,k}$ construit les tests pour une taille k et ne considère pas les données de taille inférieure. Il est alors possible de définir un critère C_2 ayant le même fonctionnement que C_1 sauf pour l'ensemble $Sub_{k,s}$. Ce critère de sélection partitionne le jeu de tests de départ, il n'y a plus de problème de chevauchement entre les sous-jeux de tests.

5.2.5 Conclusion

Dans ce chapitre, nous avons vu comment formaliser une manière de tester un programme à partir de sa spécification. Nous avons également pu formaliser la notion de critère de sélection. Les critères de sélection que nous allons étudier dans le prochain chapitre rentrent complètement dans ce cadre. Leur particularité est d'utiliser la structure des axiomes de la spécification pour effectuer le découpage. La qualité attendue pour ces critères pourra être étudiée grâce aux différentes propriétés de la section précédente (correction, complétude, partitionnement).

Chapitre 6

Une méthode de sélection par dépliage des axiomes

Dans ce chapitre, nous allons décrire deux critères de sélection de tests à partir de spécifications algébriques. Les fondements de cette méthode de test ont été donnés dans le chapitre 5. Nous voulons donc décrire des critères de sélection qui découpent le domaine des tests de manière pertinente par une méthode de sélection de tests par partition.

6.1 Introduction

Le dépliage est un critère de sélection qui utilise la structure des axiomes de la spécification pour faire ce découpage. Cette méthode peut être appliquée plusieurs fois de suite. Ainsi, après un premier découpage, il est tout à fait possible de découper de nouveau les sous-domaines, et ainsi de suite. La figure suivante donne une idée de quelques étapes dépliage, où chaque élément de surface schématise un domaine, ou un sous-domaine de test.

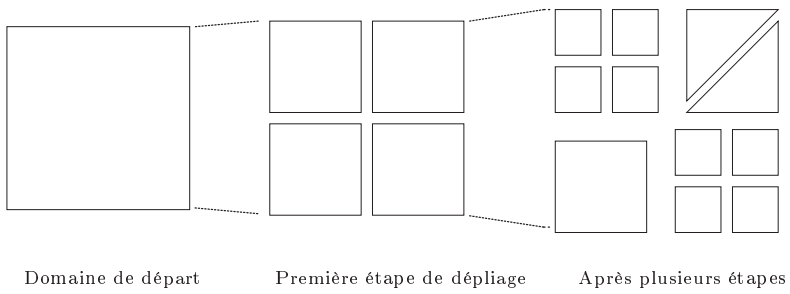


FIG. 6.1 – Exemple de dépliage

Dans les sections suivantes, nous allons décrire différentes procédure de dépliage pour différentes formes de spécifications. Nous nous intéresserons aussi spécialement à prouver que nos procédures de dépliage sont correctes, c'est-à-dire qu'elles ne construisent pas des tests qui n'existaient pas dans le domaine de test de départ, et complètes, c'est-à-dire qu'elles n'en perdent pas en cours de procédure. En d'autres termes, à chaque étape de dépliage l'union des sous-domaines reste identique. Sur la figure 6.1, cela est schématisé par le fait qu'à chaque étape la somme des surfaces correspond au carré de départ. La correction est très importante parce que sans cette propriété on pourrait créer des tests qui n'en sont pas et donc détecter des erreurs qui n'en sont pas. La complétude n'est pas nécessaire mais plutôt souhaitable si l'on ne veut pas perdre en pouvoir de détection des erreurs au fil des étapes de dépliage.

Les spécifications considérées sont des spécifications conditionnelles positives (la présentation complète est donnée dans la section 1.3.2.1). Dans la première section, nous allons donner l'idée du dépliage sur un exemple simple de spécification conditionnelle positive. Dans les deux sections suivantes, nous allons décrire deux critères de sélection par dépliage. Le premier critère est une reformulation de la procédure de sélection de tests par dépliage qui a été implanté dans l'outil de sélection LOFT (voir section 4.4 page 113). Nous proposerons également une nouvelle preuve de correction et complétude pour ce critère, beaucoup plus simple que celle donnée dans [Mar91b], en utilisant la notion de système de réécriture conditionnelle positive présentée dans la section 2.5 page 62. Le deuxième critère est une extension du précédent pour des spécifications conditionnelles positives quelconques. La preuve de correction et de complétude de cette dernière utilise la notion de normalisation d'arbre de preuve étudiée dans le chapitre 3. Ce travail a fait l'objet d'une publication à la conférence FATES'05 [AAB+05a] et d'un rapport de recherche plus détaillé [AAB+05b].

6.2 Un exemple simple

6.2.1 Une spécification conditionnelle positive

Avant de rentrer dans les détails, considérons un exemple simple de spécification dont les axiomes sont conditionnels positifs. Soit $\Sigma_{Listes} = (S_{Listes}, F_{Listes}, V_{Listes})$ une signature telle que :

$$
\begin{aligned}
S_{Listes} &= \{entier, liste, bool\} \\
F_{Listes} &= \{0 : entier, succ : entier \rightarrow entier, \\
&\quad true : bool, false : bool, \\
&\quad [] :\rightarrow liste, \\
&\quad \_ :: \_ : entier \times liste \rightarrow liste, \\
&\quad \_ \leq \_ : entier * entier \rightarrow bool, \\
&\quad insert : entier \times liste \rightarrow liste\} \\
V_{Listes} &= \{x, y : entier, L : liste\}
\end{aligned}
$$

La spécifications $Listes$ est composé de la signature Σ_{Listes} et de l'ensemble Ax_{Listes} d'axiomes conditionnels positifs suivant qui définissent les opérations $\leq$ (inférieur ou égal) et $insert$ (insertion dans une liste triée) :

$$0 \leq x = true \tag{6.1}$$
$$succ(x) \leq 0 = false \tag{6.2}$$
$$succ(x) \leq succ(y) = x \leq y \tag{6.3}$$

$$insert(x, []) = x :: [] \tag{6.4}$$
$$x \leq y = true \Rightarrow insert(x, y :: L) = x :: y :: L \tag{6.5}$$
$$x \leq y = false \Rightarrow insert(x, y :: L) = y :: insert(x, L) \tag{6.6}$$

6.2.2 Tester une opération de la spécification

Nous voulons vérifier que l'opération d'insertion $insert$ dans un programme sous test fait bien ce qui est écrit dans la spécification.

Pour l'opération d'insertion que l'on veut tester, les tests seront de la forme $insert(e, l) = v$, où e et l sont les entrées de test (des termes clos) que l'on veut soumettre au programme, et v le résultat attendu. On parle alors du domaine des tests pour l'opération $insert$.

6.2.3 La sélection par dépliage

Le dépliage utilise les axiomes de la spécification pour réaliser une sélection. Pour les tests de la forme $insert(e, l) = v$ on distingue, grâce aux axiomes, trois sous-ensembles de tests :

1. les tests de la forme $insert(x, []) = x :: []$, pour l'axiome (6.4),
2. les tests de la forme $insert(x, y :: L) = x :: y :: L$ avec $x \leq y$, pour l'axiome (6.5),
3. les tests de la forme $insert(x, y :: L) = y :: insert(x, L)$ avec $x > y$, pour l'axiome (6.6).

Pour caractériser ces sous-ensembles, nous utiliserons des ensembles de couples composés d'un ensemble d'équations (les **contraintes**) et d'une équation (l'**objectif de test**). Le **domaine de test** de l'opération $insert$ s'écrira :

$$(\{insert(e, l) = v\}, insert(e, l) = v)$$

Les trois sous-ensembles construits par dépliage s'écriront à l'aide des couples (contraintes, objectif de test) suivants :

1. pour l'axiome (6.4) : $(\{\}, insert(x, []) = x :: [])$
2. pour l'axiome (6.5) : $(\{x \leq y = true\}, insert(x, y :: L) = x :: y :: L)$
3. pour l'axiome (6.6) : $(\{x \leq y = false, v = y :: insert(x, L)\}, insert(x, y :: L) = v)$

La construction des sous-domaines se fait grâce à la notion d'unification (voir section 2.2 page 43). On peut déplier une contrainte $insert(x, l) = v$ si celle-ci s'unifie avec la partie gauche d'un axiome conditionnel positif de la forme $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow insert(e, t) = t'$. Par unification, nous construisons une substitution σ :

1. Unification de $insert(x, l)$ et $insert(x, [])$
   $\sigma = \{l/[]\}$
2. Unification de $insert(x, l)$ et $insert(x, y :: L)$
   $\sigma = \{l/y :: L\}$,
3. Unification $insert(x, l)$ et $insert(x, y :: L)$
   $\sigma = \{l/y :: L\}$

Seule la contrainte de l'axiome (6.6) peut être de nouveau dépliée sur $insert$. Si nous déplions cette contrainte sur les trois axiomes de $insert$ nous obtenons les trois couples suivants :

3.1 $(\{x \leq y = false\}, insert(x, y :: []) = y :: x :: [])$

3.2 $(\{x \leq y = false, x \leq z = true\}, insert(x, y :: z :: L) = y :: x :: z :: L)$

3.3 $(\{x \leq y = false, x \leq z = false, v = y :: z :: insert(x, L)\}, insert(x, y :: z :: L) = v)$

Les contraintes pour les axiomes (6.5) et (6.6) peuvent être dépliées sur l'opération $\leq$. La contrainte pour l'axiome (6.4) ne peut pas être dépliée plus. Le domaine caractérisé par le couple 1 est celui qui contient tous les tests de la forme $insert(x, []) = x :: []$ avec x un entier naturel. Comme nous l'avons vu, le sous-domaine correspondant à l'ajout d'un entier dans la liste vide (1er sous-domaine) ne peut être dépliée. L'utilisateur peut donc choisir, selon ses besoins, de couvrir directement ce sous-domaine (par uniformité) ou au contraire de poursuivre le découpage par une autre méthode (par exemple une régularité sur l'entier inséré).

Le dépliage découpe donc le domaine de test d'une opération en plusieurs sous-domaines construits à partir de la structure des axiomes. Une fois que nous avons réalisé un découpage assez fin du domaine de test, il sera possible de générer des tests pour chacun des sous-domaines construits. Nous ne détaillerons pas ici cette opération. Des outils existent, comme ceux décrits dans le chapitre 4 qui permettent de générer des tests à partir des sous-domaines.

6.3 Dépliage pour des spécifications conditionnelles positives

Dans cette section, nous allons définir nos critères de sélection pour des spécifications conditionnelles positives. Avant de présenter ces critères, nous allons commencer par donner le jeu de tests de référence sur lequel vont s'appliquer nos critères de sélection par dépliage, puis ensuite nous préciserons l'idée du fonctionnement du dépliage, et enfin nous définirons la notion de contrainte.

6.3.1 Jeu de tests de référence

À partir d'une spécification conditionnelle positive SP, l'ensemble de tests $SP^{\bullet} \cap Obs$ contient toutes les équations closes qui peuvent être déduites de SP (voir définition 5.1.2 page 123). Comme nous l'avons déjà dit, cet ensemble est très grand, souvent infini. Il peut contenir des tautologies, des formules redondantes ...

Pour débuter un processus de sélection de tests, nous avons besoin d'un jeu de tests de référence. Nous allons restreindre notre ensemble de tests aux équations closes de la forme $f(t_1, \ldots, t_n) = t_{n+1}$ où $\forall 1 \leq i \leq n+1,\ t_i \in T_\Sigma$ et $f \in F$. Le jeu de tests de référence sera donc défini de la manière suivante pour nos deux critères de sélection.

Définition 6.3.1 (Jeu de tests de référence) *Soit $SP = (\Sigma, Ax)$ une spécification où $\Sigma = (S, F, V)$ est la signature. Nous définissons l'ensemble $T_0(SP)$ de la manière suivante :*

$$T_0(SP) = \{ f(u_1, \ldots, u_n) = v \quad | \quad \begin{array}{l} f \in F, u_1, \ldots, u_n, v \in T_\Sigma, \\ SP \vdash f(u_1, \ldots, u_n) = v \end{array} \}$$

Exemple 6.3.1 *Pour l'opération $insert$, la formule $insert(2, 3 :: 4 :: []) = 2 :: 3 :: 4 :: []$ fait partie du jeu de tests de référence. Par contre $2 \leq 3 = true \Rightarrow insert(2, 3 :: []) = 2 :: 3 :: []$ n'en est pas un car ce n'est pas une équation, et $insert(x, []) = x :: []$ non plus car x est une variable.*

Nous pouvons remarquer que cet ensemble de formule $T_0(SP)$ correspond exactement à l'ensemble des tests que nous avons donnée dans la définition 5.1.2. En effet, l'ensemble des formules observables Obs contient toutes les équations closes qui sont des conséquences de la spécification. La principale différence réside dans le fait que l'ensemble $T_0(SP)$ est défini à l'aide du calcul $\vdash$ pour les spécification conditionnelles positives, alors que l'ensemble $SP^\bullet$ est défini sémantiquement à l'aide de la relation $\models$. L'égalité $T_0(SP) = SP^\bullet \cap Obs$ est vraie parce que le calcul conditionnel positif est correct est complet.

L'ensemble $T_0(SP)$ contient tous les tests qui sont des équations closes de Obs conséquences de la spécification SP. Cela représente encore beaucoup de tests, et surtout cela ne sélectionne rien du tout. L'idée est donc, pour commencer, de considérer les sous-jeux de tests de $T_0(SP)$ construits à partir d'une opération. Par exemple tous les tests de la forme $f(t_1, \ldots, t_n) = t_{n+1}$ pour f une opération donnée de SP, et les t_i des termes clos. Ce sous-jeu de tests est le domaine de test d'une opération.

Définition 6.3.2 *Soit $SP = (\Sigma, Ax)$ une spécification. Soit $f : s_1 \times \cdots \times s_n \rightarrow s$ une opération de la signature Σ. Le* **domaine** *de f, noté $T_0(SP)|_f$, est l'ensemble défini de la manière suivante :*

$$T_0(SP)|_f = \{ f(u_1, \ldots, u_n) = v \mid f(u_1, \ldots, u_n) = v \in T_0(SP) \}$$

L'ensemble des tests défini par $T_0(SP)|_f$ est encore trop général pour permettre de sélectionner des tests pertinent. En effet, le critère d'uniformité appliqué à un tel jeu de tests ne permettrait pas de réaliser une couverture des cas possibles.

6.3.2 Le modus ponens

L'idée du dépliage est d'utiliser la structure des axiomes de la spécification conditionnelle positive pour construire des sous-jeux de tests. Les axiomes sont de la forme $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n \Rightarrow t_{n+1} = t'_{n+1}$. Ils sont construits à partir d'un certain nombre de prémisses. Pour sélectionner des tests (qui sont des équations closes) par dépliage, il nous faudra donc éliminer ces prémisses. Pour cela nous allons utiliser la règle de modus-ponens du calcul conditionnel positif présenté dans la section 1.3.2.1 page 33) :

$$\frac{A \wedge \mathbf{u} = \mathbf{v} \Rightarrow \alpha \qquad A \Rightarrow \mathbf{u} = \mathbf{v}}{A \Rightarrow \alpha}\ M.P$$

avec A une conjonction d'équations $\bigwedge_{1 \leq i \leq m} \alpha_i$.

Pour un axiome de la forme $\alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow f(t_1, \ldots, t_n) = t$ la règle de modus-ponens permet "d'éliminer" les prémisses quand il existe une preuve de celles-ci. Une fois toutes les prémisses éliminées il ne reste qu'une équation à la racine. Cette équation n'est vérifiée que si toutes les prémisses sont vérifiées.

$$\dfrac{\alpha_m \qquad \dfrac{\alpha_{m-1} \qquad \dfrac{\alpha_2 \qquad \dfrac{\alpha_1 \qquad \dfrac{}{\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_m \Rightarrow f(t_1, \ldots, t_n) = t}\ Axiom}{\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_m \Rightarrow f(\sigma(t_1), \ldots, \sigma(t_n)) = \sigma(t)}\ Subst}{\alpha_2 \wedge \cdots \wedge \alpha_{m-1} \wedge \alpha_m \Rightarrow f(\sigma(t_1), \ldots, \sigma(t_n)) = \sigma(t)}\ M.P}{\vdots}\ M.P}{\alpha_m \Rightarrow f(\sigma(t_1), \ldots, \sigma(t_n)) = \sigma(t)}\ M.P}{f(\sigma(t_1), \ldots, \sigma(t_n)) = \sigma(t)}\ M.P$$

Les équations qui ont un arbre de preuve de cette forme correspondent à une classe particulière de tests $f(\sigma(t_1), \ldots, \sigma(t_n)) = v$ associée à cet axiome. Nous pouvons ainsi construire l'ensemble de tests pour un axiome :

$$\{f(\sigma(t_1) \ldots, \sigma(t_n)) = \sigma(t) \mid \sigma : V \to T_\Sigma, \forall \varepsilon \in \{\alpha_i\}_{i \in 1..m},\ SP \vdash \sigma(\varepsilon)\}$$

Les prémisses sont des sortes de contraintes qui doivent être vérifiées pour la substitution construite sur l'un des axiomes de la spécification. L'équation de la conclusion de l'axiome, $f(t_1, \ldots, t_n) = t$, nous donne la forme des tests à construire (objectif de test) et les prémisses les contraintes à vérifier.

6.3.3 Les contraintes

Plus généralement, nous définissons ces contraintes ainsi :

Définition 6.3.3 (Σ-Contrainte) *Un ensemble de de Σ-équations est appelé* **ensemble de Σ-contraintes.**

À partir d'un ensemble de Σ-contraintes et d'un objectif de test représenté par une équation ayant pour opérateur de tête une opération f, nous construisons un jeu de tests de la manière suivante :

Définition 6.3.4 (Ensemble des tests pour les opérations) *Soit $SP = (\Sigma, Ax)$ une spécification conditionnelle positive où $\Sigma = (S, F)$ est la signature. Soit $\mathcal{C}$ un ensemble de Σ-contraintes. Soit $f : s_1 \times \cdots \times s_n \rightarrow s$ une opération de la signature Σ, et $f(t_1, \ldots, t_n) = t$ une équation avec $t_1, \ldots, t_n, t$ des termes de $T_\Sigma(V)$.*

L'ensemble de tests pour l'opération f et pour la contrainte $\mathcal{C}$, noté $T(\mathcal{C}, f(t_1, \ldots, t_n) = t)$, est l'ensemble d'équations closes défini de la manière suivante :

$$T(\mathcal{C}, f(t_1, \ldots, t_n) = t) = \{f(\sigma(t_1) \ldots, \sigma(t_n)) = \sigma(t) \mid \begin{array}{l} \sigma : V \rightarrow T_\Sigma, \\ \forall \varepsilon \in \mathcal{C} \;\; SP \models \sigma(\varepsilon)\} \end{array}$$

Remarquons que l'ensemble $T(\{f(t_1, \ldots, t_n) = t\}, f(t_1, \ldots, t_n) = t)$, où les t_i et t sont des variables, correspond au jeu de test de référence $T_0(SP)|_f$.

Exemple 6.3.2 *Prenons l'exemple de l'axiome $x \leq y = true \Rightarrow insert(x, y :: L) = x :: y :: L$ vu précédemment. L'ensemble des tests pour cet axiome peut être caractérisé par l'ensemble $T(\{x \leq y = true\}, insert(x, y :: L) = x :: y :: L)$.*

Les ensembles de tests pour les opérations s'étendent naturellement aux ensembles de couples (contraintes, objectif de test) de la manière suivante :
Soit Γ un ensemble de couples (Σ-contraintes, Σ-équation) :

$$T(\Gamma) = \bigcup_{(\mathcal{C}, f(t_1, \ldots, t_n) = t) \in \Gamma} T(\mathcal{C}, f(t_1, \ldots, t_n) = t)$$

Les procédures de dépliage que nous allons définir dans les deux prochaines sections prennent en argument un jeu de test de référence $T_0(SP)|_f$ pour une opération f de la spécification. Avec notre notation, ce jeu de tests s'écrit :

$$T(\{(\{f(t_1, \ldots, t_n) = v\}, f(t_1, \ldots, t_n) = v)\})$$

6.3.4 Dépliage

Dans cette section nous allons détailler notre procédure de dépliage qui est une reformulation de celle de l'outil LOFT [Mar91b, Mar91a] (description détaillée de l'outil page 113).

6.3.4.1 Restrictions sur les spécifications

Dans notre procédure de dépliage, les axiomes des spécifications sont écrits d'une manière bien précise de la même manière que dans l'outil LOFT (voir section 4.4.1 page 113). Les signatures $\Sigma = (S, F, V)$ sont des signatures avec constructeurs c'est-à-dire que l'on désigne dans l'ensemble F des noms d'opérations un sous-ensemble C de constructeurs. Les équations de la conclusion des axiomes sont de la forme $g(u_1, \ldots, u_n) = v$ telles que $g \in F \setminus C$, $u_i \in T_\Omega(V)$[1] avec $\Omega = (S, C, V)$, et toutes les opérations qui ne sont pas des constructeurs dans v sont toutes plus petites que g selon un ordre bien-fondé $>_F$ construit sur F. La conclusion $g(u_1, \ldots, u_n) = v$ peut donc être orientée de gauche à droite (i.e. $g(u_1, \ldots, u_n) \rightarrow v$) comme un règle de réécriture.

Nous allons donc présenter les spécifications sous la forme d'un système de réécriture conditionnelle $\mathcal{R}$ (voir section 2.5 page 62), tel que tous les axiomes sont des règles de réécriture de la forme

$$\bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) \rightarrow v$$

avec $v, v_1, \ldots, v_n \in T_\Sigma(V)$ et $g \in F \setminus C$.

Exemple 6.3.3 *Avec de telles restrictions sur les spécifications, il est facile de transformer une spécification conditionnelle positive en un système de réécriture équivalent en orientant chaque conclusion d'axiomes de gauche à droite. Pour l'exemple de la spécification de l'insertion, nous obtenons ainsi le système de réécriture conditionnelle suivant :*

$$insert(x, []) \rightarrow x :: [] \tag{6.7}$$

$$x \leq y = true \;\Rightarrow\; insert(x, y :: L) \rightarrow x :: y :: L \tag{6.8}$$

$$x \leq y = false \;\Rightarrow\; insert(x, y :: L) \rightarrow y :: insert(x, L) \tag{6.9}$$

Cette manière de représenter les spécifications nous permettra d'assurer la correction et la complétude de la procédure de dépliage par rapport au jeu de test de référence $T_0(SP)$, dans le cas où le système de réécriture conditionnelle $\mathcal{R}$ est confluent et réducteur (voir définitions 2.3.6 page 49 et 2.5.3 page 64).

Exemple 6.3.4 *Grâce à l'ordre récursif sur les chemins $\succ^{rpo}$ résultant de l'ordre de priorité : $insert \succ \_ \leq \_ \succ \_ :: \_ \succ [] \succ false \sim true$, le système de réécriture de la spécification de l'insertion est réducteur.*

En effet,

[1] L'ensemble $T_\Omega(V)$ contient tous les termes construits uniquement sur les constructeurs, c'est-à-dire à l'aide de symboles contenus dans C.

- $insert(x, []) \succ^{rpo} x :: []$ *car* $insert \succ \_ :: \_$
- $insert(x, y :: L) \succ^{rpo} x :: y :: L$ *pour la même raison, et* $insert(x, y :: L) \succ^{rpo} x \leq y \succ^{rpo} true$
- $insert(x, y :: L) \succ^{rpo} y :: insert(x, L)$ *car* $insert \succ \_ :: \_$, *et* $\{\{x, y :: L\}\} \succ\!\!\succ^{rpo} \{\{x, L\}\}$, *et* $insert(x, y :: L) \succ^{rpo} x \leq y \succ^{rpo} false$

La confluence est évidente dans le cas de ce système de réécriture. À chaque étape, il n'y aura qu'au plus un seul axiome applicable quelque soit la formule à réécrire.

6.3.4.2 La procédure de dépliage

Notre procédure de dépliage prend donc en entrée :

- une spécification conditionnelle positive $SP = (\Sigma, Ax)$ présentée sous la forme d'un système de réécriture conditionnelle réducteur et confluent, et
- un ensemble Γ de couples composés d'un ensemble de Σ-contraintes, et d'une Σ-équation qui est l'objectif de test.

Pour garantir la correction et la complétude du critère de dépliage, le premier ensemble de Σ-contraintes que l'on considère est $\Gamma_0 = \{(\{f(x_1, \ldots, x_n) = y\}, f(x_1, \ldots, x_n) = y)\}$ où $x_i, y \in V$ $(1 \leq i \leq n)$, et $f \in F$ l'opération que l'on veut tester. Dans la suite, nos dépliages sont construits à partir de l'ensemble des tests T_{Γ_0} qui correspond au jeu de tests de référence $T_0(SP)|_f$ (voir définition 6.3.1).

La procédure de dépliage est construite comme un système formel où les formules sont des ensembles de couples (Σ-contraintes, objectif de test), et à l'aide des deux *règles d'inférences* suivantes :

1. $$\frac{\Gamma \cup \{(\mathcal{C} \cup \{r = r'\}, f(t_1, ..., t_n) = t)\}}{\Gamma \cup (\sigma(\mathcal{C}), \sigma(f(t_1, ..., t_n) = t))}\ \textbf{Réduction}$$

   avec σ l'unificateur le plus général (mgu[2]) des deux termes $r, r' \in T_\Omega(V)$ (construits sur les constructeurs uniquement).

   Cette règle a pour rôle de nettoyer nos couples construits par dépliage en simplifiant les contraintes qui peuvent être unifiées et qui ne pourront plus être dépliée puisqu'elles ne contiennent aucune opération définie.

2. $$\frac{\Gamma \cup \{(\mathcal{C} \cup \{r = s\}, f(t_1, ..., t_n) = t)\}}{\Gamma \cup \bigcup_{(c,\sigma) \in Tr(u_{|\omega}, r=s)} \{(\sigma(\mathcal{C}) \cup c, \sigma(f(t_1, ..., t_n) = t)\}}\ \textbf{Dépliage}$$

[2] Voir section 2.2 pour la définition de mgu

avec ω une position dans $u \in \{r, s\}$ et $Tr(u_{|\omega}, r = s)$, pour r et s que l'on ne peut pas unifier sur les constructeurs, l'ensemble de couples défini par :

$$\left\{ (\{\sigma(r[v]_\omega) = \sigma(s), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}, \sigma) \;\middle|\; \begin{array}{l} \sigma \text{ mgu de } u_{|\omega} \text{ et } g(v_1, \ldots, v_n), \\ \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) \rightarrow v \in Ax \end{array} \right\}$$

Cette règle réalise le découpage par dépliage des axiomes en transformant un domaine de test caractérisé par un couple (contraintes, objectif de tests) en un ensemble de sous-domaines caractérisés également par des couples. Comme la définition de $Tr(u_{|\omega}, r = s)$ est basée sur la relation de sous-terme et sur l'unification, l'ensemble est calculable si le nombre d'axiomes de la spécification SP est fini.

Le fonctionnement de la procédure de dépliage est le suivant : on prend un ensemble de couple (Σ-contraintes, objectif de test), et $\varepsilon : r = s$ une équation qui est la contrainte à déplier. Le **critère de sélection** C_ε, qui correspond à une étape de la procédure de dépliage, transforme l'ensemble :

$$\Gamma \cup \{(\mathcal{C} \cup \{r = s\}, f(t_1, \ldots, t_n) = t)\}$$

en un ensemble de couples

$$\Gamma \cup \{((\mathcal{C} \setminus \{r = s\}) \cup c, \sigma(f(t_1, \ldots, t_n) = t))\}_{(c,\sigma) \in Tr(u_{|\omega}, r=s)}$$

On notera $\Gamma \vdash_D \Gamma'$ pour indiquer que l'ensemble de de couple (Σ-contraintes, objectif de test) Γ peut être transformé en Γ' en appliquant l'une des règles de dépliage.

Exemple 6.3.1 *Reprenons l'exemple de l'insertion dans une liste triée. Grâce à cette procédure nous allons pouvoir construire un arbre de dépliage de la manière suivante :*

$$\dfrac{\dfrac{\{(\{insert(x,l) = v\}, insert(x,l) = v)\}}{\left\{\begin{array}{c} (\{v = x :: []\}, insert(x, []) = v), \\ (\{x \leq y = true, v = x :: y :: L\}, insert(x, y :: L) = v), \\ (\{x \leq y = false, v = y :: insert(x, L)\}, insert(x, y :: L) = v) \end{array}\right\}}\ \textbf{Dépliage}}{\left\{\begin{array}{c} (\{\}, insert(x, []) = x :: []), \\ (\{x \leq y = true\}, insert(x, y :: L) = x :: y :: L), \\ (\{x \leq y = false, v = y :: insert(x, L)\}, insert(x, y :: L) = v) \end{array}\right\}}\ \textbf{Réduction } (\times 2)$$

La procédure transforme le couple caractérisant l'ensemble de tests de référence de l'opération $insert$, en un ensemble de couples correspondant aux sous-domaines construits par dépliage des axiomes de la spécification. L'étape de réduction permet de simplifier les deux premières contraintes à l'aide de l'unificateur le plus général pour les contraintes de la forme $v = t$ où t un terme construit uniquement sur les constructeurs.

Pour garantir le fonctionnement correct de notre critère de sélection, nous devons éviter, à chaque étape de dépliage, que les variables rentrent en conflit. Nous devons vérifier la **condition** suivante : pour tout Γ résultant de la procédure de dépliage, pour tout $(\mathcal{C}, \Psi) \in \Gamma$, pour tout $\varepsilon \in \mathcal{C}$, et pour tout $\varphi \in Ax$ (les axiomes sur lesquels nous voulons déplier le couple $(\mathcal{C}, \Psi)$), on a

$$(Var(\Psi) \cup Var(\varepsilon)) \cap Var(\varphi) = \emptyset$$

Ceci peut être obtenu facilement en renommant les variables des axiomes par des variables fraîches à chaque itération de la procédure.

Ainsi, une procédure de dépliage est un programme qui accepte en entrée une spécification conditionnelle positive $SP = (\Sigma, Ax)$ (qui vérifie les restrictions données dans la section 6.3.4.1 page 139) et qui utilise les règles de dépliage données ci-dessus pour générer une séquence (finie ou infinie) de dépliages

$$\Gamma_0 \vdash_D \Gamma_1 \vdash_D \Gamma_2 \vdash_D \Gamma_3 \vdash_D \ldots$$

où $\Gamma_0 = \{(\{f(x_1, \ldots, x_n) = y\}, f(x_1, \ldots, x_n) = y)\}$ avec $x_i, y \in V$ ($1 \leq i \leq n$), et f une opération de la spécification.

6.3.4.3 Correction et complétude du critère de sélection

Dans le chapitre précédent, page 129, nous avons vu qu'il est souhaitable de garantir certaines propriétés sur les critères de sélection. La correction est la propriété la plus importante puisque sans elle la jeu de tests construit par découpage risque de comporter des tests qui n'en sont pas. La complétude est également une propriété importante, puisque celle-ci nous garantie que le jeu de test obtenu après application d'une étape du critère de sélection est le même que celui de départ ($|C(T)| = T$).

Nous allons maintenant montrer la correction et la complétude de cette procédure de dépliage. Cette preuve est plus simple que la preuve originale donnée dans [Mar91b] qui est très longue et contient beaucoup d'étapes intermédiaires.

Théorème 6.3.1 *Soient SP une spécification conditionnelle positive présentée sous la forme d'un système de réécriture confluent et réducteur $\mathcal{R}$, f une opération*

définie de la spécification SP, et $\Gamma_0 \vdash_D \Gamma_1 \vdash_D \Gamma_2 \vdash_D \cdots \vdash_D \Gamma_n$, l'application de n étapes de dépliage, avec $\Gamma_0 = \{(\{f(t_1, \ldots, t_n) = t\}, f(t_1, \ldots, t_n) = t)\}$.

Toutes les étapes de dépliage sont **correctes et complètes**,

$$\forall 0 \leq i \leq n, T(\Gamma_i) = T(\Gamma_{i+1}).$$

Preuve

La correction et la complétude pour la règle **Réduction** est évidente. Cette règle ne fait que simplifier l'écriture des contraintes en unifiant celles qui sont de la forme $r = r'$ avec r et r' des termes construits uniquement sur les constructeurs.

Pour la règle **Dépliage**, la preuve est la suivante :

– *(Correction)* $T_{\Gamma',f} \subseteq T_{\Gamma,f}$
Par définition de la règle d'inférence, cela revient à montrer

$$\forall c \in Tr(u_{|\omega}, t = r),\ T_{c,f} \neq \emptyset \Longrightarrow T_{c,f} \subseteq T_{\{t=r\},f}$$

pour tout $\mathcal{C} \in \Gamma$ et pour tout $t = r \in \mathcal{C}$. Soit $c' \in Tr(u_{|\omega}, t = r)$ tel que $T_{c',f} \neq \emptyset$. On suppose, en s'inspirant de la transformation de règle de dépliage, que

$$c' = \{\sigma(t[v]_\omega) = \sigma(r), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}$$

avec $t_{|\omega} = g(u_1, \ldots, u_n)$, et $\bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) \rightarrow v \in Ax$, et puis une substitution σ telle que $SP \vdash \sigma(t[v]_\omega) = \sigma(r), \forall 1 \leq i \leq m$, et $SP \vdash \sigma(\alpha_i)$ et telle que σ est aussi l'unificateur de $g(v_1, \ldots, v_n)$ et $g(u_1, \ldots, u_n)$,.
Comme SP est présentée sont la forme d'un système de réécriture réducteur et confluent, nous obtenons directement $\sigma(t[g(v_1, \ldots, v_n)]_\omega) \rightarrow \sigma(t[v]_\omega)$. De plus, par la propriété de confluence, nous avons $\sigma(t[v]_\omega) \overset{*}{\leftrightarrow} \sigma(r)$, et donc $\sigma(t[g(v_1, \ldots, v_n)]_\omega) \overset{*}{\leftrightarrow} \sigma(r)$.
Grâce à l'unification dans σ, nous avons aussi $\sigma(u_i) \overset{*}{\leftrightarrow} \sigma(v_i)$ et donc $\sigma(g(u_1, \ldots, u_n)) \overset{*}{\leftrightarrow} \sigma(g(v_1, \ldots, v_n))$ d'où $\sigma(t) \overset{*}{\leftrightarrow} \sigma(t[g(v_1, \ldots, v_n]_\omega])$.

Nous pouvons donc en conclure que $\sigma(t) \overset{*}{\leftrightarrow} \sigma(r)$ et donc $SP \vdash t = r$ en utilisant simplement les contraintes de notre c' quelconque.

– *(Complétude)* $T_{\Gamma,f} \subseteq T_{\Gamma',f}$
Pour montrer la complétude, on suppose que Γ a été transformé en Γ' à partir d'une contrainte $t = r \in \Gamma$ et une position $\omega \in t$. Comme le système de réécriture conditionnelle qui représente SP est confluent, nous avons : $SP \vdash \sigma(t) = \sigma(r) \Longleftrightarrow \sigma(t) \overset{*}{\leftrightarrow} \sigma(r)$. On a nécessairement, pour $t_{|\omega} = g(u_1, \ldots, u_n)$, un axiome $\bigwedge_{1 \leq i \leq m} t_i = t'_i \Rightarrow g(v_1, \ldots, v_n) \rightarrow v$ dans SP et une

substitution close $\rho : V \to T_\Sigma$ tels que $\sigma(t_{|\omega}) = \rho(g(v_1, \ldots, v_n))$, et donc, par la confluence SP, $\sigma(t) \to_\mathcal{R} \sigma(t[\rho(g(v_1, \ldots, v_n))]_\omega) \overset{*}{\leftrightarrow}_\mathcal{R} \sigma(r)$ et pour tout $1 \leq i \leq m$, $\rho(t_i) \overset{*}{\leftrightarrow}_\mathcal{R} \rho(t'_i)$. Comme $Var(t = r) \cap Var(\bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) = v) = \emptyset$, il y a une unique substitution close σ' telle que $\sigma'(t) = \sigma(t)$, $\sigma'(r) = \sigma(r)$, $\sigma'(g(v_1, \ldots, v_n)) = \rho(g(v_1, \ldots, v_n))$, $\sigma'(v) = \rho(v)$ et pour tout i, $\sigma'(t_i = t'_i) = \rho(t_i = t'_i)$. Par conséquent, $\sigma'(t) = \sigma'(t[g(v_1, \ldots, v_n)]_\omega)$, et donc σ' est un unificateur de $t_{|\omega}$ et $g(v_1, \ldots, v_n)$. Donc pour tout $1 \leq i \leq m$, nous avons $\sigma'(u_i) \overset{*}{\leftrightarrow}_\mathcal{R} \sigma'(v_i)$.

★

Grâce au théorème 6.3.1, nous avons prouvé la correction et la complétude de cette première procédure de sélection de tests. Ainsi, un découpage peut être réalisé par l'utilisateur où il peut guider le processus de dépliage comme il le souhaite, tout en sachant qu'il conservera intact le pouvoir de détection d'erreurs de son jeu de tests de référence.

6.3.5 Dépliage étendu

6.3.5.1 Cadre de la procédure

L'un des inconvénients majeur de la procédure de dépliage que nous venons d'étudier est que les spécifications conditionnelles positives doivent respecter des restrictions fortes (voir section 6.3.4.1 page 139). Ces restrictions permettent de garantir la correction et la complétude de la procédure. Cependant, de telles restrictions réduisent considérablement le pouvoir d'expression des formules conditionnelles positives.

Exemple 6.3.2 *Considérons les axiomes suivants pour définir les opérations $first$, $reverse$ et $last$ sur les listes :*

$$first(x :: l) = x \quad (6.10)$$
$$reverse([]) = [] \quad (6.11)$$
$$reverse(reverse(l)) = l \quad (6.12)$$
$$last(l) = first(reverse(l)) \quad (6.13)$$

Nous appellerons SP_a la spécification composée des 4 axiomes précédents et

de la signature $\Sigma_a = (S_a, F_a, V_a)$ *telle que :*

$$\begin{aligned} S_a &= \{entier, liste\} \\ F_a &= \{0 : entier, succ : entier \rightarrow entier, \\ &\quad [] :\rightarrow liste, \_ :: \_ : entier \times liste \rightarrow liste, \\ &\quad first : liste \rightarrow entier, last : liste \rightarrow entier, \\ &\quad reverse : liste \rightarrow liste\} \\ V_a &= \{x : entier, l : liste\} \end{aligned}$$

Les quatre axiomes de SP_a *sont des équations, ils rentrent donc bien dans le cadre des axiomes conditionnels positifs. Cependant, ils ne respectent les restrictions du premier critère de sélection par dépliage (voir section 6.3.4.1). L'axiome pour* $first$ *n'est pas défini dans le cas de la liste vide. Le deuxième axiome pour* $reverse$ *n'est pas constructif : il décrit une propriété abstraite sur les listes mais ne permet pas de construire les listes qui correspondent. L'unique axiome pour* $last$ *est également abstrait puisqu'il ne définit l'opération* $last$ *qu'à partir d'autres opérations qui elles-mêmes ne vérifient pas les restrictions que nous avons donné.*

Nous allons maintenant décrire une nouvelle procédure de dépliage étendu qui prend en compte des spécifications quelconques (comme celle donnée en exemple). La seule contrainte requise, pour des résultats semblables de correction et de complétude, est que les spécifications sont conditionnelles positives. Les axiomes ne sont plus orientés comme des règles de réécriture conditionnelles. Cependant, cette liberté à l'étape de spécification rend plus complexe la procédure de dépliage étendu.

6.3.5.2 La procédure de dépliage étendu

Comme dans la section précédente, la procédure de dépliage prend en entrée :
- une spécification conditionnelle positive $SP = (\Sigma, Ax)$ (sans aucune autre contrainte), et
- un ensemble Γ de couples (Σ-contraintes, objectif de test).

Le premier ensemble considéré, correspondant au jeu de test de référence, sera également $\Gamma_0 = \{(\{f(x_1, \ldots, x_n) = y\}, f(x_1, \ldots, x_n) = y)\}$ où $x_i, y \in V$ $(1 \leq i \leq n)$.

Exemple 6.3.3 *Pour l'opération* $first$ *dont les axiomes sont donnés dans l'exemple 6.3.2, nous pouvons poser l'ensemble de couples (contraintes-objectif de test) suivant :*

$$\Gamma_{first} = \{(\{first(L) = e\}, first(L) = e)\}$$

avec e et L des variables de type $entier$ et $liste$ respectivement. Ce couple caractérise tous les tests de la forme $first(l) = x$ avec x et l des termes clos tels que $first(x) = l$ est une conséquence de la spécification SP_a.

Le fonctionnement de ce dépliage est similaire à celui présenté dans la section précédente. Le dépliage tel que nous l'avons étudié précédemment est un sous cas de ce dépliage étendu. La principale différence de fonctionnement réside dans le fait que l'on ne déplie pas seulement sur des axiomes de la forme $\bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow g(v_1,...,v_n)=v$, c'est-à-dire où l'opération dépliable se trouve en tête de la partie gauche de la conclusion d'un axiome. Le dépliage étendu s'effectue pour tous les axiomes dont la partie droite ou gauche peut s'unifier avec un sous-terme des contraintes.

La procédure de **dépliage étendu** est définie par les deux règles d'inférence suivantes :

$$\frac{\Gamma \cup \{(\mathcal{C} \cup \{r = r'\}, f(t_1, ..., t_n) = t)\}}{\Gamma \cup \{(\sigma(\mathcal{C}), \sigma(f(t_1, ..., t_n) = t))\}}\ \textbf{Réduction}$$

avec σ le mgu des deux termes $r, r' \in T_\Omega(V)$ (construits sur les constructeurs uniquement).

$$\frac{\Gamma \cup \{(\mathcal{C} \cup \{\varepsilon\}, f(t_1, ..., t_n) = t\}}{\Gamma \cup \bigcup_{(c,\sigma)\in Tr(\varepsilon)} \{(\sigma(\mathcal{C}) \cup c, \sigma(f(t_1, ..., t_n) = t))\}}\ \textbf{Dépliage}$$

où $Tr(\varepsilon)$ pour $\varepsilon = (r = s)$ (ou symétriquement $s = r$) avec r et s qui ne peuvent pas être toutes les deux des variables, est l'ensemble de Σ-contraintes défini ainsi :

$$\left\{ \left\{ (\{\sigma(r[v]_\omega) = \sigma(s), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}, \sigma) \;\middle|\; \begin{array}{c} \sigma \text{ mgu de } r_{|\omega} \text{ et } g(v_1, \ldots, v_n), \\ (\bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow g(v_1, \ldots, v_n) = v \in Ax \\ ou \\ \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow v = g(v_1, \ldots, v_n) \in Ax) \end{array} \right\} \right.$$
$$\cup$$
$$\left. \left\{ (\{\sigma(r) = \sigma(s[v]_\omega), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}, \sigma) \;\middle|\; \begin{array}{c} \sigma \text{ mgu de } s_{|\omega} \text{ et } g(v_1, \ldots, v_n), \\ (\bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow g(v_1, \ldots, v_n) = v \in Ax \\ ou \\ \bigwedge_{1\leq i\leq m}\alpha_i \Rightarrow v = g(v_1, \ldots, v_n) \in Ax) \end{array} \right\} \right\}$$

Comme la définition de $Tr(t = r)$ est basée sur la relation de sous-terme et sur l'unification, l'ensemble est calculable si la spécification SP a un nombre fini d'axiomes.

Le fonctionnement de la procédure est le suivant : soit une équation ε, le critère de sélection C_ε transforme l'ensemble $\Gamma \cup \{(\mathcal{C} \cup \{\varepsilon\}, f(t_1, ..., t_n) = t)\}$ caractérisant un jeu de tests, en un ensemble de couples :

$$\Gamma \cup \{((\mathcal{C} \setminus \{\varepsilon\}) \cup c, \sigma(f(t_1, ..., t_n) = t)\}_{(c,\sigma)\in Tr(\varepsilon)}$$

On notera $\Gamma \vdash_{D_e} \Gamma'$ pour indiquer que Γ peut être transformé en Γ' en appliquant l'une des règles de dépliage étendu.

Exemple 6.3.4 *Nous allons déplier sur l'exemple de l'opération $first$ pour lequel nous avons donné l'ensemble $\Gamma_{first} = \{(\{first(L) = e\}, first(L) = e)\}$. Il n'y a qu'une seule contrainte $first(L) = e$ qui contient seulement l'opération $first$. Nous allons donc déplier par rapports aux axiomes qui la définissent c'est-à-dire les axiomes $first(x :: l) = x$ (6.10) et $last(l) = first(reverse(l))$ (6.13). En effet, pour ce dépliage, la partie droite de l'axiome (6.13) est également prise en compte, contrairement au dépliage "classique".*

Après une étape de dépliage nous obtenons les deux couples suivants :

1. *Pour l'axiome (6.10), on construit l'unificateur de L et $x :: l$ et on obtient ainsi le couple $(\{x = e\}, first(x :: l) = e)$ qui une fois simplifié à l'aide de la règle de Réduction caractérise tous les tests de la forme $first(x :: l) = x$ avec x et l des termes clos. L'ensemble de contraintes ne contient plus d'opérations définies de la spécification, il n'est donc plus possible de déplier.*
2. *Pour l'axiome (6.13), on construit l'unificateur de L et $reverse(l)$ et on obtient ainsi le couple $(\{last(l) = e\}, first(reverse(l)) = e)$ qui caractérise tous les tests de la forme $first(reverse(l)) = e$ avec $SP \models last(l) = e$ et l un terme clos tels que $first(reverse(l)) = e$ est conséquence de la spécification. L'ensemble de contraintes contient l'opération $last$ qu'il est possible de déplier à nouveau grâce à l'axiome 6.13.*

Le deuxième couple est dépliable à nouveau de la manière suivante :

2.1. Le dépliage sur l'axiome (6.13) donne le couple $(\{first(reverse(l)) = e\}, first(reverse(l)) = e)$ qui caractérise tous les tests de la forme $first(reverse(l)) = e$ tels que $SP \models e = first(reverse(l))$ avec l un terme clos tel que l'équation est une conséquence de la spécification. Il y a a maintenant deux opérations définies dans l'ensemble des contraintes, et l'on doit effectuer le dépliage sur tous les axiomes possibles :

2.1.1. Pour l'axiome (6.10)*, il n'est pas possible d'unifier* $x :: l$ *et* $reverse(l)$*, le dépliage n'est donc pas possible.*

2.1.2 Pour (6.11)*, on obtient le couple* $(\{first([]) = e\}, first(reverse([])) = e)$ *qui caractérise le test* $first(reverse([])) = first([])$*. Il n'est donc évidemment plus possible de déplier ici.*

2.1.3 Pour (6.12)*, le couple* $(\{first(l) = e\}, first(reverse(reverse(l))) = e)$ *caractérise tous les tests de la forme* $first(reverse(reverse(l))) = e$ *avec* $SP \models e = first(l)$*. Il serait possible de déplier à nouveau la contrainte sur l'opération* $first$*.*

2.1.4 Pour (6.13)*, on revient sur le deuxième couple de la première étape de dépliage* $(\{last(l) = e\}, first(reverse(l)) = e)$ *qui caractérise donc les mêmes tests.*

Nous pouvons observer que la procédure de dépliage décrite ci-dessus est fortement combinatoire. C'est le résultat d'un dépliage complet sur tous les sous termes des contraintes. Ceci assure la complétude de la procédure par rapport au jeu de tests de référence $T_0(SP)$ (voir section suivante).

L'intérêt de cette procédure de dépliage est qu'elle peut être appliquée à n'importe quelle spécification conditionnelle positive qui a un nombre fini d'axiomes. Aucune autre condition n'est imposée pour assurer la correction et la complétude. Par conséquent, cette procédure nous permet de faire du test fonctionnel à un niveau plus abstrait que dans le cadre des spécifications "exécutables".

6.3.5.3 Correction et complétude du critère de sélection

Comme précédemment, pour le résultat de complétude de notre première procédure de dépliage, nous allons montrer que le critère de sélection du dépliage étendu est complet. Il est également nécessaire de renommer les variables par des variables fraîches à chaque étape de dépliage.

Théorème 6.3.2 *Si* $\Gamma \vdash_{D_e} \Gamma'$ *alors* $T(\Gamma) = T(\Gamma')$.

Preuve

Le cas de la règle **Réduction** est évident.

Pour la règle **Dépliage**, la preuve est la suivante :

– $(Correction)\ T(\Gamma') \subseteq T(\Gamma)$.

Par définition de la règle d'inférence cela revient à montrer

$$\forall (c, \sigma) \in Tr(t = r),\ T_{c,f(\sigma(u_1),\ldots,\sigma(u_n))=\sigma(v)} \neq \emptyset \Longrightarrow T_{c,f(\sigma(u_1),\ldots,\sigma(u_n))=v} \subseteq T_{\{t=r\},f(u_1,\ldots,u_n)=v}$$

pour tout $(\mathcal{C}, f(u_1, \ldots, u_n) = v) \in \Gamma$ et pour tout $t = r \in \mathcal{C}$.

Donc, soit $(c, \sigma) \in Tr(t = r)$ tel que $T_{c,f} \neq \emptyset$. Sans perte de généralité, nous supposons que

$$c = \{\sigma(t[v]_\omega) = \sigma(r), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}$$

avec $t_{|_\omega} = g(v_1, \ldots, v_n)$, et $\bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) = v \in Ax$ et puis $\rho : V \to T_\Sigma$ une substitution close, tels que $SP \vdash \rho(\sigma(t[v]_\omega)) = \rho(\sigma(r))$ et $\forall 1 \leq i \leq m$, $SP \vdash \rho(\sigma(\alpha_i))$. L'unification nous assure que $SP \vdash \sigma(u_1) = \sigma(v_1), \ldots, SP \vdash \sigma(u_n) = \sigma(v_n)$.
Nous pouvons construire l'arbre de preuve :

$$\dfrac{\dfrac{\mathcal{A}}{\rho(\sigma(t)) = \rho(\sigma(t[v]_\omega))}\, Trans \quad \dfrac{\vdots}{\rho(\sigma(t[v]_\omega)) = \rho(\sigma(r))}}{\rho(\sigma(t)) = \rho(\sigma(r))}\, Trans$$

avec $\mathcal{A}$ le sous-arbre de preuve suivant :

$$\dfrac{\dfrac{\rho(\sigma(u_1))=\rho(\sigma(v_1))\ldots\rho(\sigma(u_n))=\rho(\sigma(v_n))}{\vdots}}{\rho(\sigma(t))=\rho(\sigma(t[g(v_1,\ldots,v_n)]_\omega)} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha_1\wedge\ldots\wedge\alpha_n\Rightarrow g(v_1,\ldots,v_n)=v}{\vdots}}{\alpha_1\wedge\ldots\wedge\alpha_m\Rightarrow t[g(v_1,\ldots,v_n)]_\omega=t[v]_\omega} \quad \dfrac{\vdots}{\rho(\sigma(\alpha_1))}}{\rho(\sigma(\alpha_1))\wedge\ldots\wedge\rho(\sigma(\alpha_m))\Rightarrow\rho(\sigma(t[g(v_1,\ldots,v_n)]_\omega))=\rho(\sigma(t[v]_\omega))} \quad \dfrac{\vdots}{\rho(\sigma(\alpha_2))}}{\rho(\sigma(\alpha_2))\wedge\ldots\wedge\rho(\sigma(\alpha_m))\Rightarrow\rho(\sigma(t[g(v_1,\ldots,v_n)]_\omega))=\rho(\sigma(t[v]_\omega))}}{\rho(\sigma(\alpha_3))\wedge\ldots\wedge\rho(\sigma(\alpha_m))\Rightarrow\rho(\sigma(t[g(v_1,\ldots,v_n)]_\omega))=\rho(\sigma(t[v]_\omega))} \\ \dfrac{\vdots}{\rho(\sigma(\alpha_m))\Rightarrow\rho(\sigma(t[g(v_1,\ldots,v_n)]_\omega))=\rho(\sigma(t[v]_\omega))} \quad \dfrac{\vdots}{\rho(\sigma(\alpha_m))} \\ \rho(\sigma(t[g(v_1,\ldots,v_n)]_\omega))=\rho(\sigma(t[v]_\omega))$$

$$\rho(\sigma(t)) = \rho(\sigma(t[v]_\omega))$$

Cela signifie donc que l'on prouve $SP \vdash \rho(\sigma(t)) = \rho(\sigma(r))$ en se servant uniquement de la contrainte c.

– *(Complétude)* $T_{\Gamma,f} \subseteq T_{\Gamma',f}$.
Par définition de la règle d'inférence, cela revient à montrer

$$T_{\{t=r\},f} \subseteq \bigcup_{c \in Tr(t=r)} T_{c,f}$$

Ceci est équivalent à montrer que pour toute substitution close $\sigma : V \to T_\Sigma$ telle que $SP \vdash \sigma(t) = \sigma(r)$, il existe $c \in Tr(t = r)$ telle que pour toute $\varepsilon \in c,\ SP \vdash \sigma(\varepsilon)$.

On remarque que cette procédure de dépliage définit une stratégie qui limite l'espace de recherche pour les arbres de preuve à une classe donnée d'arbres qui ont un forme spécifique. Par conséquent, la procédure de dépliage définit un stratégie qui sélectionne des arbres de preuves (voir les règles de la section 1.3.2.1 page 33) où

– aucune instance de la transitivité n'apparaît au-dessus des instances des règles de symétrie, substitution, remplacement, et modus-ponens seulement quand la transitivité apparaît sur la prémisse gauche du modus-ponens,
– aucune instance de modus-ponens n'apparaît au-dessus des instances de symétrie, substitution, et remplacement,
– aucune instance de symétrie et remplacement n'apparaît au-dessus de la substitution.

L'inclusion ci-dessus, est prouvée en montrant qu'il y a un arbre de preuve, de l'énoncé $SP \vdash \sigma(t) = \sigma(r)$ satisfaisant la structure spécifique décrite ci-dessus. Pour cela, nous allons utiliser le résultat du chapitre 3 qui consiste à définir des transformations élémentaires d'arbre de preuve, et à montrer ainsi que la transformation globale termine.
Voici les règles nécessaires pour cette transformation :

– La $substitution$ remonte sur la **transitivité**

$$\dfrac{\dfrac{\Gamma \Rightarrow t = u \quad \Gamma \Rightarrow u = v}{\Gamma \Rightarrow t = v}\ Trans}{\sigma(\Gamma) \Rightarrow \sigma(t) = \sigma(v)}\ Subst \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma \Rightarrow t = u}{\sigma(\Gamma) \Rightarrow \sigma(t) = \sigma(u)}\ Subst \quad \dfrac{\Gamma \Rightarrow u = v}{\sigma(\Gamma) \Rightarrow \sigma(u) = \sigma(v)}\ Subst}{\sigma(\Gamma) \Rightarrow \sigma(t) = \sigma(v)}\ Trans$$

– La *substitution* remonte sur le **modus-ponens**

$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow \beta \quad \Gamma \Rightarrow \alpha}{\Gamma \Rightarrow \beta}\ MP}{\sigma(\Gamma) \Rightarrow \sigma(\beta)}\ Subst \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow \beta}{\sigma(\Gamma) \wedge \sigma(\alpha) \Rightarrow \sigma(\beta)}\ Subst \quad \dfrac{\Gamma \Rightarrow \alpha}{\sigma(\Gamma) \Rightarrow \sigma(\alpha)}\ Subst}{\sigma(\Gamma) \Rightarrow \sigma(\beta)}\ MP$$

– La règle de $substitution$ remonte sur la règle de **symétrie**

$$\dfrac{\dfrac{\Gamma \Rightarrow t_1 = t_2}{\Gamma \Rightarrow t_2 = t_1}\ Sym}{\sigma(\Gamma) \Rightarrow \sigma(t_2) = \sigma(t_1)}\ Subst \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\Gamma \Rightarrow t_1 = t_2}{\sigma(\Gamma) \Rightarrow \sigma(t_1) = \sigma(t_2)}\ Subst}{\sigma(\Gamma) \Rightarrow \sigma(t_2) = \sigma(t_1)}\ Sym$$

– La règle de $substitution$ remonte sur la règle de **remplacement**

$$\dfrac{\dfrac{\Gamma \Rightarrow t_i = t'_i}{\Gamma \Rightarrow f(t_1,\ldots,t_i,\ldots,t_n) = f(t_1,\ldots,t'_i,\ldots,t_n)}\ Rempl}{\sigma(\Gamma) \Rightarrow f(\sigma(t_1),\ldots,\sigma(t_i),\ldots,\sigma(t_n)) = f(\sigma(t_1),\ldots,\sigma(t'_i),\ldots,\sigma(t_n))}\ Subst$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\Gamma \Rightarrow t_i = t'_i}{\sigma(\Gamma) \Rightarrow \sigma(t_i) = \sigma(t'_i)}\ Subst}{\sigma(\Gamma) \Rightarrow f(\sigma(t_1),\ldots,\sigma(t_i),\ldots,\sigma(t_n)) = f(\sigma(t_1),\ldots,\sigma(t'_i),\ldots,\sigma(t_n))}\ Rempl$$

– La règle de *remplacement* remonte sur la règle de **transitivité**

$$\dfrac{\dfrac{\Gamma \Rightarrow t_i = t'_i \qquad \Gamma \Rightarrow t'_i = t''_i}{\Gamma \Rightarrow t_i = t''_i}\ Trans}{\Gamma \Rightarrow f(\ldots t_i \ldots) = f(\ldots t''_i \ldots)}\ Rempl \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma \Rightarrow t_i = t'_i}{\Gamma \Rightarrow f(\ldots t_i \ldots) = f(\ldots t'_i \ldots)}\ Rempl \quad \dfrac{\Gamma \Rightarrow t'_i = t''_i}{\Gamma \Rightarrow f(\ldots t'_i \ldots) = f(\ldots t''_i \ldots)}\ Rempl}{\Gamma \Rightarrow f(\ldots t_i \ldots) = f(\ldots t''_i \ldots)}\ Trans$$

– La règle de *remplacement* remonte sur la règle de **modus ponens**

$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow t_i = t'_i \qquad \Lambda \Rightarrow \alpha}{\Gamma \wedge \Lambda \Rightarrow t_i = t'_i}\ MP}{\Gamma \wedge \Lambda \Rightarrow f(t_1,\ldots,t_i,\ldots,t_n) = f(t_1,\ldots,t'_i,\ldots,t_n)}\ Rempl \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow t_i = t'_i}{\Gamma \wedge \alpha \Rightarrow f(t_1,\ldots,t_i,\ldots,t_n) = f(t_1,\ldots,t'_i,\ldots,t_n)}\ Rempl \qquad \Lambda \Rightarrow \alpha}{\Gamma \wedge \Lambda \Rightarrow f(t_1,\ldots,t_i,\ldots,t_n) = f(t_1,\ldots,t'_i,\ldots,t_n)}\ MP$$

– La règle de *symétrie* remonte sur la règle de **transitivité**

$$\dfrac{\dfrac{\Gamma \Rightarrow t_2 = t_3 \qquad \Gamma \Rightarrow t_3 = t_1}{\dfrac{\Gamma \Rightarrow t_2 = t_1}{\Gamma \Rightarrow t_1 = t_2}\ Sym}\ Trans}{} \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma \Rightarrow t_3 = t_1}{\Gamma \Rightarrow t_1 = t_3}\ Sym \quad \dfrac{\Gamma \Rightarrow t_3 = t_3}{\Gamma \Rightarrow t_3 = t_2}\ Sym}{\Gamma \Rightarrow t_1 = t_2}\ Trans$$

– La règle de *symétrie* remonte sur la règle de **modus ponens**

$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow t_1 = t_2 \qquad \Gamma \Rightarrow \alpha}{\Gamma \Rightarrow t_1 = t_2}\, MP}{\Gamma \Rightarrow t_2 = t_1}\, Sym \qquad \rightsquigarrow$$

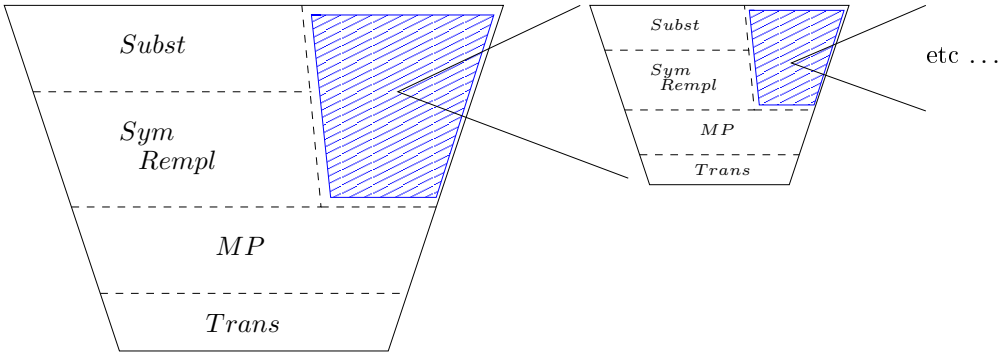
$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow t_1 = t_2}{\Gamma \wedge \alpha \Rightarrow t_2 = t_1}\, Sym \qquad \Gamma \Rightarrow \alpha}{\Gamma \Rightarrow t_2 = t_1}\, MP$$

– La règle de *modus-ponens* remonte sur la règle de **transitivité** (lorsque celle-ci se trouve sur la partie gauche du modus-ponens)

$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow t_1 = t_2 \qquad \Gamma \wedge \alpha \Rightarrow t_2 = t_3}{\Gamma \wedge \alpha \Rightarrow t_1 = t_3}\, Trans \qquad \Gamma \Rightarrow \alpha}{\Gamma \Rightarrow t_1 = t_3}\, MP \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\Gamma \wedge \alpha \Rightarrow t_1 = t_2 \qquad \Gamma \Rightarrow \alpha}{\Gamma \Rightarrow t_1 = t_2}\, MP \qquad \dfrac{\Gamma \wedge \alpha \Rightarrow t_2 = t_3 \qquad \Gamma \Rightarrow \alpha}{\Gamma \Rightarrow t_2 = t_3}\, MP}{\Gamma \Rightarrow t_1 = t_3}\, Trans$$

Nous remarquons que les transformations élémentaires d'arbres de preuve effectuent une sorte de "distribution" des instances de substitutions au-dessus des autres règles, de symétrie et remplacement au-dessus de transitivité et modus-ponens, et modus-ponens au-dessus de transitivité.
Nous obtenons un arbre de la forme suivante :



Les parties hachurées représentent les arbres qui se trouvent au-dessus de la partie droite des règles de modus-ponens (ce sont en fait les preuves des prémisses éliminées par le modus-ponens).

Nous pouvons définir un ordre récursif sur les chemins (recursive path ordering) $>^{rpo}$ pour ordonner les instances de règles du calcul conditionnel positif à l'aide de la relation de priorité suivante

$$Subst > Sym \sim Rempl > MP > Trans$$

on montre que $\overset{*}{\rightsquigarrow} \subseteq >^{rpo}$ et donc que $\rightsquigarrow$ termine [3].

Par conséquent, pour n'importe quel énoncé $SP \vdash \sigma(t) = \sigma(r)$, si $t = r$ n'est pas une tautologie (i.e. de la forme $u = u$), il y a nécessairement un axiome $\bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) = v$, une position ω dans $\sigma(t)$ ou $\sigma(r)$ et une substitution close ρ tels que soit $\sigma(t)_{|\omega} = \rho(g(v_1, \ldots, v_n))$ ou $\sigma(r)_{|\omega} = \rho(g(v_1, \ldots, v_n))$. Comme $Var(t = r) \cap Var(\bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) = v) = \emptyset$, il y a une unique substitution close σ' telle que $\sigma'(t) = \sigma(t)$, $\sigma'(r) = \sigma(r)$, $\sigma'(g(v_1, \ldots, v_n)) = \rho(g(v_1, \ldots, v_n))$, $\sigma'(v) = \rho(v)$ et pour tout i, $\sigma'(\alpha_i) = \rho(\alpha_i)$. Par conséquent, $\sigma'(t) = \sigma'(t[g(v_1, \ldots, v_n)]_\omega)$ ou $\sigma'(r) = \sigma'(t[g(v_1, \ldots, v_n)]_\omega)$, et donc σ' est un unificateur de $t_{|\omega}$ ou $r_{|\omega}$ et $g(v_1, \ldots, v_n)$. Donc, par notre transformation globale d'arbre de preuve, il existe nécessairement un arbre de preuve associé à l'énoncé $SP \vdash \sigma(t) = \sigma(r)$ de la forme suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow g(v_1,\ldots,v_n)=v}{\sigma'(\alpha_1) \wedge \ldots \wedge \sigma'(\alpha_m) \Rightarrow \sigma'(g(v_1,\ldots,v_n))=\sigma'(v)} \quad \dfrac{\vdots}{\sigma'(\alpha_1)}}{\sigma'(\alpha_2) \wedge \ldots \wedge \sigma'(\alpha_m) \Rightarrow \sigma'(g(v_1,\ldots,v_n))=\sigma'(v)} \quad \dfrac{\vdots}{\sigma'(\alpha_2)}}{\vdots} \quad \dfrac{\vdots}{\sigma'(\alpha_m)}}{\sigma'(\alpha_m) \Rightarrow \sigma'(g(v_1,\ldots,v_n))=\sigma'(v)}}{\sigma'(g(v_1,\ldots,v_n))=\sigma'(v)}}{\vdots}}{\sigma'(t)=\sigma'(t[v]_\omega)} \quad \dfrac{\vdots}{\sigma'(t[v]_\omega)=\sigma'(r)}}{\sigma(t) = \sigma(r)}$$

Donc, nous avons $c = \{\sigma'(t[v]_\omega) = \sigma'(r), \sigma'(\alpha_1), \ldots, \sigma'(\alpha_m)\}$.

★

[3] $\overset{*}{\rightsquigarrow}$ est la fermeture transitive et réflexive de $\rightsquigarrow$.

6.4 Conclusion

Le dépliage permet de réaliser une couverture de tous les sous-cas possibles à partir des axiomes. Cependant, les couples qu'il construit caractérisent des sous-domaines de tests, mais pas des tests concrets. En pratique, afin de pouvoir tester un programme, il faut ensuite générer des tests par uniformité ou régularité (voir section 5.2.1). À l'image des outils de sélection LOFT et HOL-TestGen (voir chapitre 4), il est tout à fait possible de générer automatiquement et aléatoirement des tests à partir des contraintes construites par dépliage. Ainsi, on peut obtenir des jeux de tests concrets de taille raisonnable mais qui couvrent de manière judicieuse l'ensemble de tous les tests possibles.

Conclusion et Perspectives

Nous avons présenté dans cette thèse des résultats faisant partie de deux domaines importants en vérification formelle : la preuve et le test. Les résultats que nous avons donnés sur la normalisation d'arbres de preuve, nous ont permis de prouver la complétude de notre critère de sélection de tests par dépliage.

Après une première partie composée de rappels théoriques, la deuxième partie est une présentation de deux résultats de normalisation d'arbres de preuve. Ces résultats généraux ont pour cadre les systèmes formels. Ils nous ont permis de garantir la terminaison de procédures de transformation d'arbres de preuve. Celles-ci fonctionnent par application successive de règles de transformation élémentaires (assimilables à des règles de réécriture) sur un arbre de preuve. Si ces règles élémentaires vérifient certaines conditions alors nous garantissons que la procédure de normalisation termine. Nous avons donné deux ensembles de conditions suffisantes pour deux résultats : le premier dit de normalisation forte et le deuxième dit de normalisation faible.

Nous avons validé cette approche en donnant de nombreux exemples d'application à des théorèmes connus : l'élimination des coupures dans le calcul des séquents, la logicalité dans la logique équationnelle, ou encore le lemme de Newman en réécriture. Les preuves de ces résultats ont pour caractéristique commune de pouvoir être exprimées sous la forme d'une procédure de transformation d'arbres de preuve définie par un ensemble de règles de transformations élémentaires. Ensuite, comme ces règles vérifient bien nos conditions suffisantes, les preuves de ces théorèmes sont fortement simplifiées. Outre la reformulation de ces résultats bien connus, nous avons validé l'apport de notre méthode de normalisation sur un nouveau résultat dans le cadre du test de logiciel. Nous avons pu ainsi prouver la complétude de notre critère de sélection par dépliage des axiomes (dans le cas de spécifications conditionnelles positives sans restrictions sur la forme des axiomes).

Dans la troisième partie de ce manuscrit, divisée en trois chapitres, nous nous intéressons à définir des critères de sélection de tests à partir de spécifications algébriques.

– Dans le chapitre 4, nous avons donné un bref état de l'art pour le test fonctionnel qui nous a amené à présenter un panel d'outils existants (LOFT [Mar95], HOL-TestGen [BW05b] et QuickCheck [CH00]).

– Dans le chapitre 5, nous avons présenté un cadre générique de test à partir de spécifications axiomatiques [LGA96]. Nous avons introduit la notion importante de critère de sélection. Cela nous a permis de formaliser deux propriétés pour les critères de sélection de tests par partition : la correction et la complétude.

– Dans le chapitre 6 nous avons donné nos deux critères particuliers de sélection de jeux de tests à partir de spécifications algébriques [AAB+05a]. La sélection se fait à partir des axiomes de la spécification. Elle consiste à faire un découpage du domaine de tous les tests possibles par dépliage des axiomes. Un premier critère a été donné pour le cas de spécifications conditionnelles positives "exécutables". Les principes de ce dépliage sont les mêmes que dans l'outil LOFT, mais nous avons reformulé le critère dans notre cadre générique et simplifié la preuve de correction et de complétude. La classe de spécifications considérée pour ce critère peut paraître trop restreinte. Nous avons donc généralisé le dépliage pour des spécifications conditionnelles positives abstraites (sans autres restrictions sur la forme des axiomes). C'est une nouveauté par rapport aux outils d'aide à la sélection de tests qui utilisent toujours des spécifications exécutables. Finalement, nous avons pu donner la preuve de correction et de complétude pour ce critère de sélection de tests par dépliage étendu. À l'aide des résultats de normalisation de la deuxième partie de ce manuscrit nous avons pu donner la preuve de complétude. Nous avons construit pour cela des règles de transformations élémentaires d'arbres de preuve et utilisé le théorème de normalisation forte pour prouver la terminaison.

Perspectives

Nous pouvons, en guise de perspectives, nous demander s'il n'est pas possible de généraliser ces critères de sélection à des classes plus larges de spécifications. Y aura-t-il toujours un moyen de garantir des propriétés intéressantes (correction et complétude) sur les critères de sélection ? Pour cela, il faudra sûrement commencer par trouver une classe de spécifications plus seulement conditionnelles positives mais vérifiant un certain nombre de conditions (un peu comme notre première procédure pour des spécifications conditionnelles exécutables). À priori, l'utilisation du calcul des séquents et de la règle de coupure (au lieu du modus ponens pour les formules conditionnelles) est une perspective intéressante.

Récemment, les travaux de Delphine Longuet ont permis d'étendre les critères de sélection par dépliage à d'autres formalismes : pour la logique du premier ordre sans les quantificateurs [AAGL07] et pour la logique modale [LA07].

Sous une autre approche, nous pouvons également citer les spécifications structurées [Mac00a, MS02]. Celles-ci sont construites par combinaison de plusieurs spécifications à l'aide de primitives telles que celles du langage CASL [ABK+02, CoF04]. Un objectif intéressant serait d'adapter le dépliage à de telles spécifications.

À moyen terme et à l'image des outils existants, il serait intéressant d'implanter nos stratégies de sélection de tests. L'intégration de celles-ci dans HOL-TestGen [BW05b, BW05c] (présenté dans la section 4.2) permettrait de bénéficier de toutes les fonctionnalités de cet outil pour tester un programme. En effet, le dépliage tel que nous l'avons présenté permet de guider la sélection des sous-domaines, ce que ne permet pas la stratégie de HOL-TestGen.

Pour notre cadre de normalisation d'arbres de preuve, l'intégration à un assistant de preuve comme Isabelle serait envisageable. Il serait également très utile de confronter nos conditions pour la normalisation forte et faible à d'autres résultats de normalisation en logique (notamment l'élimination des coupures présente dans de nombreux formalismes).

Bibliographie

[AAB+05a] Marc AIGUIER, Agnès ARNOULD, Clément BOIN, Pascale Le GALL et Bruno MARRE : Testing from algebraic specifications : Test data set selection by unfolding axioms. *In Formal Approaches to Testing of Software (FATES'05)*, LNCS. Springer-Verlag, juillet 2005.

[AAB+05b] Marc AIGUIER, Agnès ARNOULD, Clément BOIN, Pascale Le GALL et Bruno MARRE : Testing from algebraic specifications : Test data set selection by unfolding axioms. Rapport technique 110-2005, LaMI - CNRS et Université d'Evry Val d'Essonne, 2005.

[AAGL07] Marc AIGUIER, Agnès ARNOULD, Pascale Le GALL et Delphine LONGUET : Test selection criteria for quantifier-free first order specifications. *In Fundamental of Software Engineering*, volume à paraître de *LNCS*. Springer, 2007.

[ABD02] Marc AIGUIER, Diane BAHRAMI et Catherine DUBOIS : On a generalised logicality theorem. *In AISC'2002*, volume 2385 de *L.N.A.I.*, pages 51–64. Springer Verlag, 2002.

[ABG98] Marc AIGUIER, Gille BERNOT et Pascale Le GALL : Fondements logiques pour les méthodes formelles. Cours de DEA, Université d'Évry Val d'Essonne, 1998.

[ABK+02] Egidio ASTESIANO, Michel BIDOIT, Hélène KIRCHNER, Bernd KRIEG-BRÜCKNER, Peter D. MOSSES, Donald SANNELLA et Andrzej TARLECKI : Casl : the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[ABL05] Marc AIGUIER, Clément BOIN et Delphine LONGUET : On a generalized theorem for normalization of proof trees. Rapport technique, LaMI - CNRS et Université d'Evry Val d'Essonne, 2005.

[Abr96] J. R. ABRIAL : *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Aig02] Marc AIGUIER : Sémantique des langages de programmation. Cours de maîtrise d'informatique, Université d'Évry Val d'Essonne, 2002.

[AKKB99] Egidio ASTESIANO, H. J KREOWSKI et B. KRIEG-BRUCKNER : *Algebraic Foundations of Systems Specification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[ALG02] Agnès ARNOULD et Pascale LE GALL : Test de conformité : une approche algébrique. *Technique et Science Informatiques, Test de logiciel, vol. 21, n°9*, pages 1219–1242, 2002.

[ALGM96] Agnès ARNOULD, Pascale LE GALL et Bruno MARRE : Dynamic testing from bounded data type specifications. *In Dependable Computing - EDCC-2*, volume 1150 de *L.N.C.S*, pages 285–302, 1996.

[Arn97] Agnès ARNOULD : *Test à partir de spécifications de structures bornées : une théorie du test, un méthode de sélection, un outil d'assistance à la sélection*. Thèse, Université de Paris XI - Orsay, 1997.

[Bah03] Diane BAHRAMI : *Une axiomatisation de la réécriture abstraite*. Thèse de doctorat, LaMI - Université d'Évry Val d'Essonne, 2003.

[BBG97] Gilles BERNOT, Laurent BOUAZIZ et Pascale Le GALL : A theory of probabilistic functional testing. *In ICSE '97 : Proceedings of the 19th international conference on Software engineering*, pages 216–226. ACM Press, 1997.

[BCFG86] L. BOUGÉ ;, N. CHOQUET, L. FRIBOURG et M.C. GAUDEL : Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, 1986.

[Bei90] Boris BEIZER : *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BGM91] G. BERNOT, M.-C. GAUDEL et B. MARRE : Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[Bir35] G. BIRKHOFF : On the structure of abstract algebras. *In Proceedings of The Cambridge Philosophical Society*, volume 31, pages 433–454, 1935.

[BM04] Michel BIDOIT et Peter D. MOSSES : CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[BN98] F. BAADER et T. NIPKOW : *Term Rewriting and All That*. C.U. Press, 1998.

[Boi03] Clément BOIN : Un résultat général de normalisation d'arbres de preuve. Rapport de stage de DEA, LaMI - Université d'Evry Val d'Essonne, 2003.

[BW05a] Achim D. BRUCKER et Burkhart WOLFF : HOL-TestGen 1.1.0 user guide. Rapport technique 482, ETH Zürich, avril 2005.

[BW05b] Achim D. BRUCKER et Burkhart WOLFF : Interactive testing using HOL-TestGen. *In* Wolfgang GRIESKAMP et Carsten WEISE, éditeurs : *Formal Approaches to Testing of Software*, Lecture Notes in Computer Science. Springer-Verlag, Edinburgh, 2005.

[BW05c] Achim D. BRUCKER et Burkhart WOLFF : Symbolic test case generation for primitive recursive functions. *In* Jens GRABOWSKI et Brian NIELSEN, éditeurs : *Formal Approaches to Testing of Software*, numéro 3395 de Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, Linz, 2005.

[BW07] Achim D. BRUCKER, et Burkhart WOLFF : Using hol-testgen for test-sequence generation with an application to firewall testing. *In TAP 2007 : Tests And Proofs*, Lecture Notes in Computer Science. Springer-Verlag, Zurich, 2007. A paraître.

[CH00] Koen CLAESSEN et John HUGHES : Quickcheck : a lightweight tool for random testing of haskell programs. *In ICFP '00 : Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[CoF04] COFI (THE COMMON FRAMEWORK INITIATIVE) : CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[Der82] N. DERSHOWITZ : Orderings for term rewriting systems. *In Theoretical Computer Science*, volume 17, pages 279–301, 1982.

[DF93] J. DICK et A. FAIVRE : Automating the generation and sequencing of test cases from model-based specifications. *In FME'93 : Industial-Strenth Formal Methods, First International Symposium of Formal Methods Europe*, volume 670 de *LNCS*, pages 268–284, Odense, Denmark, April 1993. Springer Verlag.

[DG03] J. DAWSON et R. GORE : A new machine-checked proof of strong normalisation for display logic. *In* James HARLAND, éditeur : *Electronic Notes in Theoretical Computer Science*, volume 78. Elsevier, 2003.

[DJ90] N. DERSHOWITZ et J.P. JOUANNAUD : *Rewrite systems*, pages 243–320. MIT Press, Cambridge, MA, USA, 1990.

[DW99] G. DOWEK et B. WERNER : Proof normalization modulo. *In Types for proofs and programs 98*, volume 1657 de *L.N.C.S*, pages 62–67, 1999.

[DW00] M. DOCHE et V. WIELS : Extended institutions for testing. *In AMAST'2000*, numéro 1816 de Lecture Notes in Computer Science, pages 514–528, 2000.

[ECR06] ECRC : *ECLIPSE 5.10, ECLIPSE Common Logic Programming System, User Manual*. Disponible à l'adresse http ://eclipse.crosscoreop.com, 2006.

[Gal86] J.-H. GALLIER : *Logic for Computer Science*, volume Foundations of Automatic Theorem Proving. Harper & Row, 1986.

[Gau95] M.C. GAUDEL : Testing can be formal, too. *In TAPSOFT'95, International Joint Conference, Theory And Practice of Software Development*, volume 915 de *LNCS*, pages 82–96, Aarhus, Denmark, 1995. Springer Verlag.

[Gen35] G. GENTZEN : Untersuchungen über das logische schlieβen. *Mathematische Zeitschrift*, 39(176-210):405–431, 1935. English translation in M.-E. Szabo, editor, The collected Papers of Gerhard Gentzen, pages 68-131, North-Holland, 1969.

[GG75] John B. GOODENOUGH et Susan L. GERHART : Toward a theory of test data selection. *In Proceedings of the international conference on Reliable software*, pages 493–510, New York, NY, USA, 1975. ACM Press.

[GMSB96] M.C. GAUDEL, B. MARRE, F. SCHLIENGER et G. BERNOT : *Précis de génie logiciel*. Masson, 1996.

[Kap84] S. KAPLAN : Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984.

[KATP02] Pieter W. M. KOOPMAN, Artem ALIMARINE, Jan TRETMANS et Marinus J. PLASMEIJER : Gast : Generic automated software testing. *In IFL*, pages 84–100, 2002.

[KK06] Claude KIRCHNER et Hélène KIRCHNER : *Rewriting, Solving, Proving*. Version Préliminaire, 2006.

[Kle52] S.-C. KLEENE : *Introduction to Meta-mathematics*. North-Holland, 1952.

[Klo92] J.W. KLOP : Term rewriting systems. *In Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[LA07] Delphine LONGUET et Marc AIGUIER : Specification-based testing for cocasl's modal specifications. *In Conference on Algebra and Coalgebra in Computer Science*, volume à paraître de *LNCS*. Springer, 2007.

[Lal90] René LALEMENT : *Logique, réduction, résolution*. Masson, 1990.

[LG93] P. LE GALL : *Les algèbres étiquetées : une sémantique pour les spécifications algébriques fondée sur une utilisation systématique des termes. Application au test de logiciel avec traitement d'exceptions.* Thèse, Université de Paris XI - Orsay, 1993.

[LGA96] Pascale LE GALL et Agnès ARNOULD : Formal specification and test : correctness and oracle. *In 11th WADT joint with the 9th general COMPASS workshop*, volume 1130 de *LNCS*, pages 342–358. Springer, 1996. Oslo, Norway, Sep. 1995, Selected papers.

[LPU02] B. LEGEARD, F. PEUREUX et M. UTTING : Automated boundary testing from z and b. *In FME (Formal Methods Europe)*, volume 2391 de *LNCS*, pages 21–40. Springer-Verlag, 2002.

[LY96] D. LEE et M. YANNAKAKIS : Principles and methods of testing finite state machines - A survey. *In Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[MA00] Bruno MARRE et Agnès ARNOULD : Test sequences generation from LUSTRE descriptions : GATEL. *In ASE-00 : The 15th IEEE Conference on Automated Software Engineering*, pages 229–237, Grenoble, septembre 2000. IEEE CS Press.

[Mac00a] Patrícia D. L. MACHADO : Testing from structured algebraic specifications. *In AMAST2000*, volume 1816 de *LNCS*, pages 529–544, 2000.

[Mac00b] Patrícia D. L. MACHADO : *Testing from Structured Algebraic Specifications : The Oracle Problem*. Thèse, University of Edinburgh, 2000.

[Mar91a] Bruno MARRE : *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*. Thèse, Université de Paris XI - Orsay, 1991.

[Mar91b] Bruno MARRE : Toward an automatic test data set selection using algebraic specifications and logic programming. *In* K. FURUKAWA, éditeur : *Eight International Conference on Logic Programming (ICLP'91)*, pages 25–28. MIT Press, 1991.

[Mar95] Bruno MARRE : Loft : A tool for assisting selection of test data sets from algebraic specifications. *In TAPSOFT'95 : Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark*, volume 915 de *Lecture Notes in Computer Science*, pages 799–800. Springer, 1995.

[MB04] Bruno MARRE et Benjamin BLANC : Test selection strategies for lustre descriptions in gatel. *In MBT 2004 joint to ETAPS'2004*, volume 111 de *ENTCS*, pages 93–111, 2004.

[MS02] Patrícia D. L. MACHADO et Donald SANNELLA : Unit testing for casl architectural specifications. *In Mathematical Foundations of Computer Science*, LNCS, pages 506–518. Springer-Verlag, 2002.

[New42] M. H. A. NEWMAN : On theories with a combinatorial definition of equivalence. *Annals of Mathematics (2)*, 43:223–243, 1942.

[NPW02] Tobias NIPKOW, Lawrence C. PAULSON et Markus WENZEL : *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 de *LNCS*. Springer, 2002.

[Rob65] J. A. ROBINSON : A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Tai89] W.-W. TAIT : Normal derivability in classical logic. *In* J. BARWISE, éditeur : *The Syntax and Semantics of Infinitary Languages*, pages 204–236. Springer-Verlag, 1989.

[Tre92] J. TRETMANS : *A Formal Approach to Conformance Testing*. Thèse, University of Twente, 1992.

[Tre96] G. J. TRETMANS : Conformance testing with labelled transition systems : Implementation relations and test generation. *Computer networks and ISDN systems*, 29(1):49–79, 1996.

[UL07] Mark UTTING et Bruno LEGEARD : *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann, 2007.

[vO04] V. van OOSTROM : Sub-birkhoff. *In Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, volume 2998 de *LNCS*, pages 180–195, Nara, April 2004. Springer.

[Wal90] L.-A. WALLEN : *Automated Deduction for Non Classical Logics*. The MIT Press, 1990.

[Wen05] Markus WENZEL : The Isabelle/Isar Reference Manual. Rapport technique, TU München, octobre 2005.

[Wir90] Martin WIRSING : Algebraic specification. *In Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 675–788, Cambridge, MA, USA, 1990. MIT Press.

[XRK00] S. XANTHAKIS, P. RÉGNIER et C. KARAPOULIOS : *Le test des logiciels*. études et logiciels informatique. Hermes Science Publications, 2000. Paris.